Tries for Lexical Analysis

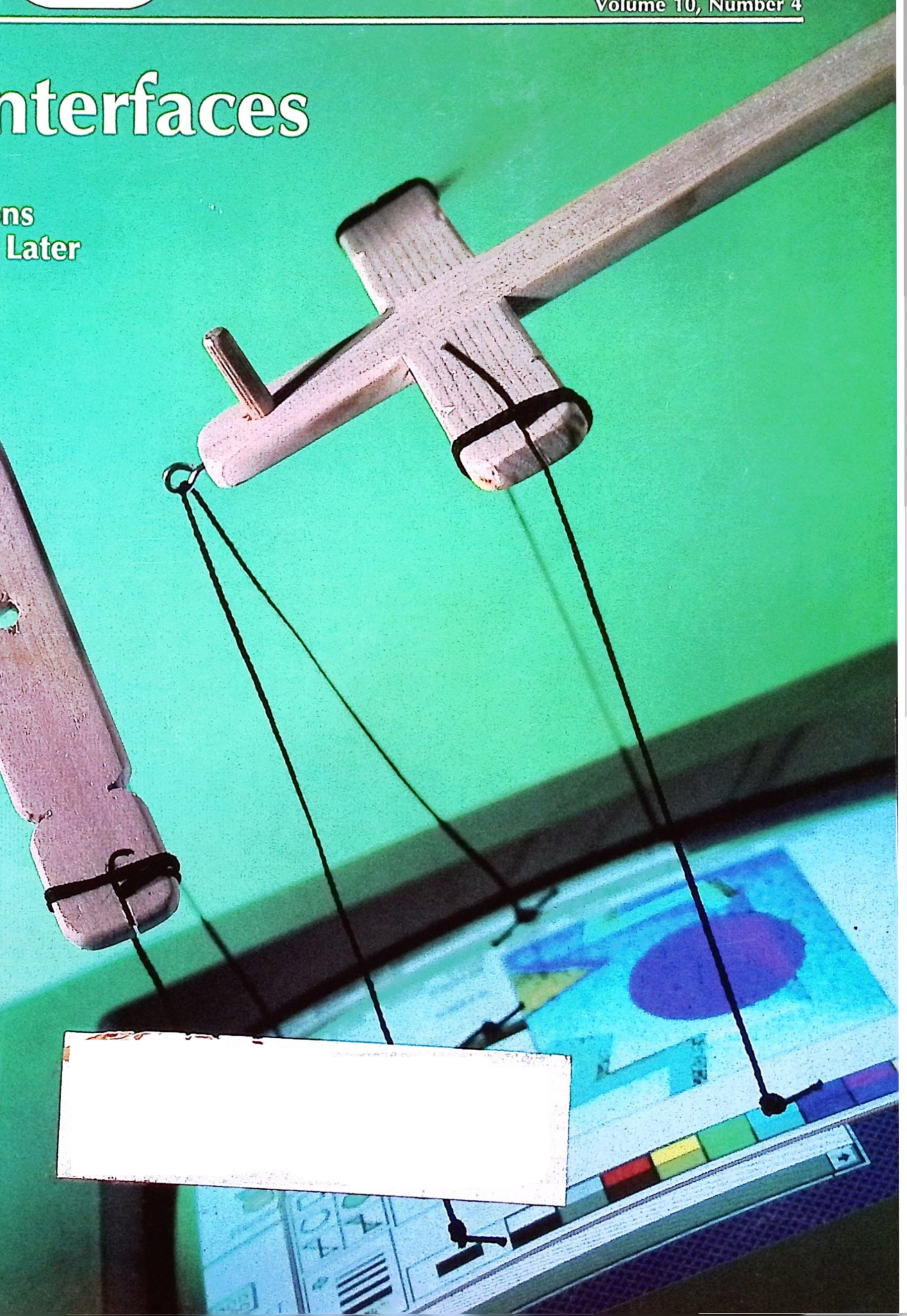
Price $4.50 U.S.
(Canada $5.95)

The C Users Journal™

April 1992 — Volume 10, Number 4

User Interfaces

- Interactive Menu Selections
- Adding a GUI Later



Also:

- A C++ Money Class
- Multi-Copy Math Functions
- User Report: Zinc Class Library

Columns By:
Rex Jaeschke
Kenji Hino
P. J. Plauger
Ken Pugh
Sydney Weinstein

04

0 38808 74098 9

April 1992
Vol. 10, No. 4

The C Users Journal™

USER INTERFACES

OTHER FEATURES

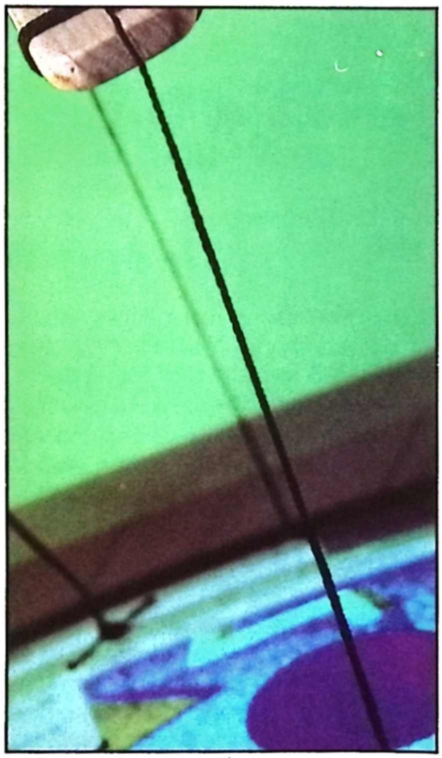


Photo by Robert Ward.

COLUMNS

USER REPORT

BOOK REVIEW

DEPARTMENTS

The C Users Journal (ISSN 0898-9788) is published monthly by R&D Publications, Inc., 2601 Iowa, Lawrence, KS 66046 (913) 841-1631. Second-class postage paid at Lawrence, KS and additional mailing offices. POSTMASTER: Send address changes to THE C USERS JOURNAL, 2601 Iowa, Lawrence, KS 66046. Subscriptions: Annual renewable subscriptions to The C Users Journal are $29.95 US, $54 Canada and Mexico, $65 overseas. Payments must be made in US dollars. Make checks payable to The C Users Journal.

WE'VE CHARTED THE WAY WITH WIN++

...for Windows applications in Borland C++

We've chosen the direction and plotted the course to easy and powerful Windows applications development!

Now with over 150 classes, Win++™ provides a simplified, flexible and powerful high-level C++ interface to make your Windows development efforts a breeze. Classes for setting internal data structures and integrating data into applications are also included.

Put the wind at your back...

Central to Win++ is that it provides a class for each type of display object in Windows. Win++ includes ready-made classes for representing dialog boxes, child controls, pens, brushes, menus, bitmaps, cursors, icons, a printer, a clipboard, and many others. **New classes support the dynamic data exchange management library (DDEML), including clients, servers, advise loops and multiple simultaneous conversations.** Win++ also supports the construction of dynamic-link libraries (DLLs). Use Win++'s ready-made DLLs in your programs and save valuable time, memory and disk space. And, you may distribute Win++'s DLLs with your applications.

It's smooth sailing from here out...

Win++ comes with a comprehensive reference manual and tutorial. Complete source code is included. You can rely on Blaise Computing to provide and support quality add-on libraries. We've been doing so since 1982 — it's our business — our only business.

Success is in the offing...

Win++ requires Windows 3.0 or later and Borland C++. Win++ costs just $289. We're so convinced that you'll find Win++ essential to your applications that if during the first 60 days you're not completely satisfied, we'll refund your money. No questions asked.

Call our order department toll free at (800) 333-8087! Or, by fax, (510) 540-1938.
Ask about training and consulting services!

BLAISE COMPUTING INC.
819 Bancroft Way Berkeley, California 94710 (510) 540-5441

◆ Request 102 on Reader Service Card ◆

Trademarks are property of their respective holders.

The C Users Journal™

EDITORIAL

Publisher	*Robert Ward*
Assistant to Publisher	*Donna Stucky Ward*
Senior Editor	*P. J. Plauger*
Managing Editor	*Diane Thomas*
Associate Editor	*Martha Masinton*
Contributing Editors	*Kenji Hino*
	Rex Jaeschke
	Kenneth Pugh
	Dan Saks
	Sydney Weinstein
	Leor Zolman

ADVERTISING AND MARKETING

Marketing Director	*Jeff Dickey-Chasins*
Acct. Manager, East	*Ed Day*
Acct. Manager, Midwest	*Donna Stucky Ward*
Acct. Manager, West	*Edwin Rothrock*
Sales Services	*Paula Cobb*
Direct Marketing	*Bill Uhler*

PRODUCTION

Art Director	*Susan Schuette Buchanan*
Graphic Artist	*Twyla Watson Bogaard*
Typographer	*Ann Bröcker*
Production Coordinator	*Liza Behymer*

CIRCULATION

Customer Service	*Chrissy Trybom*
Order Fulfillment	*Jodi Leonard*

STAFF

Business Manager	*Cherilyn Merrill*
Technical	*Kenji Hino*
	Leor Zolman

The C Users Journal is the successor to the *C Users' Group Newsletter* and *The C Journal*. Subscribers are automatically enrolled as members of The C Users Group. *The C Users Journal* and The C Users' Group are services of R&D Publications, Inc., Lawrence, KS.

The Audit Bureau

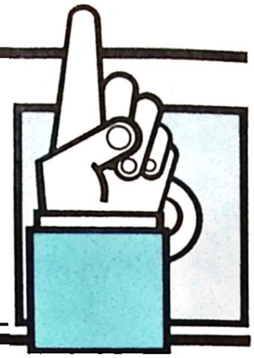
Advertising: For rate cards or other information on placing advertising in *The C Users Journal*, contact the advertising department at (913) 841-1631, or write *The C Users Journal*, 1601 W. 23rd St., Ste. 200, P.O. Box 3127, Lawrence, KS 66046-0127.

Customer Service: For subscription orders and address changes, contact *The C Users Journal*, 1601 W. 23rd St., Ste. 200, P.O. Box 3127, Lawrence, KS 66046-0127. Telephone (913) 841-1631; FAX (913) 841-2624.

Library Submissions Sought: The C Users' Group Library is the world's largest holding of public domain and Shareware C software and is composed primarily of submissions by members. C programmers are encouraged to submit software to the Library. Send software on disk (no subdirectories on MS-DOS disks please) to The C Users' Group, 1601 W. 23rd St., Ste. 200, P.O. Box 3127, Lawrence, KS 66046-0127. (913) 841-1631.

Trademarks: The C Users Journal, R&D Publications, Inc. Doctor C's Pointers, Rex Jaeschke. UNIX, AT&T Bell Laboratories. XENIX, MS-DOS, OS/2, Microsoft C and QuickC, Windows, Microsoft Corporation. IBM, IBM-PC, Micro Channel, PS2, International Business Machines Corp. Turbo C, Turbo C++, Borland International. Macintosh, Apple Computer, Inc. Zortech C++, Zortech. Roland MPU-401. VAX-VMS, Digital Equipment Corp.

Editor's Forum



I have been thinking a lot about standards lately. That should be no surprise, since I *write* about standards a lot. Still, my perspective has been shifting lately.

When I got involved in the C standard back in 1983, the world was different. Most of us compiler vendors cared intensely about forming a good standard. Many of our potential customers had a different focus. They were more concerned that we didn't mess up a language they had come to depend on. Experience with other major programming language standards was not encouraging. Delays, excess invention, even litigation seemed to be par for the course.

C helped usher in a new era. Here was a standard being formed by active representatives from a broad community. No single vendor dominated the proceedings, nor did a small clique of language inventors. Here was a standard with widespread commercial support from the outset.

And that's what has become the norm. Computing is a big and very competitive business. Complying with standards is of interest to more than just government contractors. The world can't afford bad standards any more. So everyone seems to have something to say about programming standards now.

Even more interesting, programming standards have become a serious matter in the international community. With the growth in international trade, everyone wants a level playing field. Having serious international standards helps the small players compete with the larger ones. It also helps the larger players meet the needs of the smaller markets.

Both ANSI and ISO are scrabbling to adapt to this new state of affairs. The volume of commentary and the pressure to develop timely standards keeps growing, even as the standards become more complex. I don't see how they are going to change fast enough, but I know that they must.

P.J. Plauger
`pjp@plauger.uunet`

P.J. Plauger
pjp@plauger.uunet.uu.net

Kills bugs dead.
Keeps bugs away.

Slash Development Time.
This year and next, C programmers are going to spend millions of hours looking for memory allocation errors. What a shame. Tens of thousands of product dates will slip, and hundreds of thousands of aspirin will be taken. Finally, there's a solution that doesn't involve changing a single line of code.

Meet Your New Watchdog.
MemCheck is an error detection and prevention tool that helps C developers reduce development time and meet application reliability standards. MemCheck will find memory leakage, buffer overwrites, and other nasties with reports that identify the exact location of the culprit. Quickly. Automatically. *Painlessly.*

Greater Certainty.
Whether you're bitten by bugs or just want to make sure none are "hiding," MemCheck is the easiest addition you can make to your software development environment. MemCheck integrates seamlessly with existing C code, with versions that are custom-tailored for all major compilers. Just plug and play — no hassles, no rough edges.

Industrial Strength.
Up your Quality Assurance standards. Find your toughest C bugs. Join the ranks of the major developers who have discovered MemCheck. No product adds so much value with so little effort. Find out why MemCheck is setting standards in software reliability.

MEMCHECK™

EFFORTLESS ERROR DETECTION AND PREVENTION

To order, call (313) 996-2944

Suggested retail prices: Microsoft C $139.95, Borland C $139.95, Intel Code Builder $179.95, Watcom C $179.95, Windows SDK $179.95

All major credit cards accepted. Fax orders welcome: (313) 747-8519. Please specify compiler and diskette size.

STRATOSWARE CORPORATION Suite 1500, 1756 Plymouth Road, Ann Arbor, MI 48105

Trademarks mentioned are the property of their respective owners.

◆ Request 414 on Reader Service Card ◆

P.J. Plauger

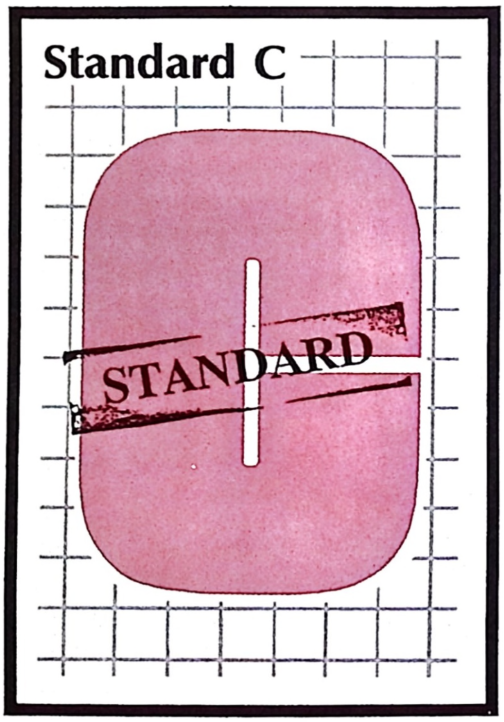
The Header *<stdlib.h>*

Introduction

The header *<stdlib.h>* is a hodgepodge. Committee X3J11 invented this header as a place to define macros and declare functions that had no other sensible home:

- Many existing functions, such as *abs* and *malloc*, had no traditional headers to declare them. X3J11 felt strongly that *every* functions should be declared in a standard header. If such a function seemed out of place in all other headers, it ended up declared in *<stdlib.h>*.
- New groups of macros and functions ended up in new standard headers wherever possible. *<float.h>* and *<locale.h>* are clear examples. Additions to existing groups ended up in existing headers. *strcoll*, declared in *<string.h>*, and *strftime*, declared in *<time.h>*, are also fairly clear. Other macros and functions are harder to categorize. These ended up defined or declared in *<stdlib.h>*.

This header is not the only hodgepodge. I discuss the evolution of the header *<stddef.h>* in *Standard C, CUJ* December 1991.



To provide some structure, I organize the functions into six groups:

- integer math (*abs*, *div*, *labs*, and *ldiv*) — performing simple integer arithmetic
- algorithms (*bsearch*, *qsort*, *rand*, and *srand*) — capturing operations complex and widespread enough to warrant packaging as library functions
- text conversions (*atof*, *atoi*, *atol*, *strtod*, *strtol*, and *strtoul*) — determining encoded arithmetic values from text representations
- multibyte conversions (*mblen*, *mbstowcs*, *mbtowc*, *wcstombs*, and *wctomb*) — mapping between multibyte and wide-character encodings
- storage allocation (*calloc*, *free*, *malloc*, and *realloc*) — managing a heap of data objects
- environmental interactions (*abort*, *atexit*, *exit*, *getenv*, and *system*) — interfacing between the program and the execution environment

I discuss separately how to implement the functions in each of these groups. This month, I cover only the first two groups. I won't bother to present the header as a whole. It's pretty straightforward.

P.J. Plauger is senior editor of The C Users Journal. *He is secretary of the ANSI C standards committee, X3J11, and convenor of the ISO C standards committee, WG14. His latest book is* The Standard C Library, *published by Prentice-Hall. You can reach him care of* The C Users Journal *or via Internet at* *pjp@plauger.uunet.uu.net*.

Listing 1 (abs.c)

```
/* abs function */
#include <stdlib.h>

int (abs)(int i)
    {   /* compute absolute value of int argument */
    return ((i < 0) ? -i : i);
    }

/* End of File */
```

Listing 2 (div.c)

```
/* div function */
#include <stdlib.h>

div_t (div)(int numer, int denom)
    {   /* compute int quotient and remainder */
    div_t val;

    val.quot = numer / denom;
    val.rem = numer - denom * val.quot;
    if (val.quot < 0 && 0 < val.rem)
        {   /* fix remainder with wrong sign */
        val.quot += 1;
        val.rem -= denom;
        }
    return (val);
    }

/* End of File */
```

Listing 3 (labs.c)

```
/* labs function */
#include <stdlib.h>

long (labs)(long i)
    {   /* compute absolute value of long argument */
    return ((i < 0) ? -i : i);
    }

/* End of File */
```

Listing 4 (ldiv.c)

```
/* ldiv function */
#include <stdlib.h>

ldiv_t (ldiv)(long numer, long denom)
    {   /* compute long quotient and remainder */
    ldiv_t val;

    val.quot = numer / denom;
    val.rem = numer - denom * val.quot;
    if (val.quot < 0 && 0 < val.rem)
        {   /* fix remainder with wrong sign */
        val.quot += 1;
        val.rem -= denom;
        }
    return (val);
    }

/* End of File */
```


The C Standard on Integer Math

7.10.6 Integer arithmetic functions

7.10.6.1 The *abs* function

Synopsis

```
#include <stdlib.h>
int abs(int j);
```

Description

The *abs* function computes the absolute value of an integer *j*. If the result cannot be represented, the behavior is undefined.[130] [FN130. The absolute value of the most negative number cannot be represented in two's complement.]

Returns

The *abs* function returns the absolute value.

7.10.6.2 The *div* function

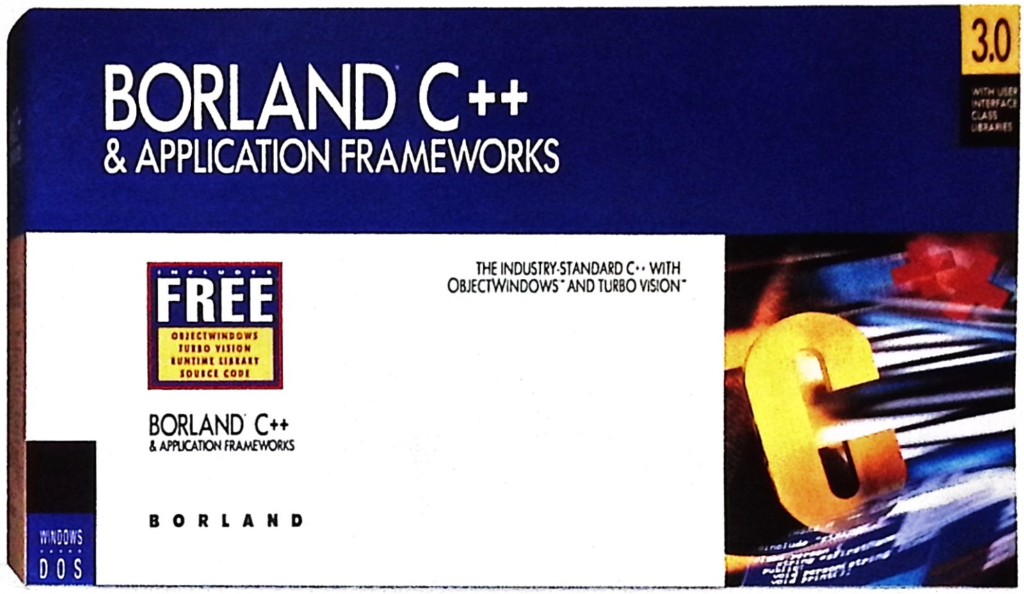
Synopsis

```
#include <stdlib.h>
div_t div(int numer, int denom);
```

Description

The *div* function computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise, *quot * denom + rem* shall equal *numer*.

NEW! VERSION 3.0



...If you program for a living

Borland® C++ & Application Frameworks 3.0 is *the* choice of professional C and C++ programmers for Windows and DOS application development. With unmatched optimizations, powerful tools, unsurpassed Windows development environment and object-oriented application frameworks, Borland C++ & Application Frameworks 3.0 has no equal. Quite simply, if you program for a living, this is everything you need.

OOP, to simplify your life

Borland C++ & Application Frameworks 3.0 simplifies programming by giving you ready-made user interface objects that plug directly into your application. Automatically inherit windows, menus, scroll bars, mouse support and more. Add an editor in just one line. With Object-Oriented Programming (OOP), you get amazing code reusability, extensibility and easier maintenance because applications are built on a base of tested, reliable code.

New features give you incredible programming options!

Just look at some of the enhanced features in Borland C++ 3.0:

- ANSI C and C++ 2.1 and templates
- Global optimizer includes:
  - Global register allocation
  - Local and global common sub-expressions
  - Induction variables
  - Loop and jump optimization
  - Register parameter passing
  - And ten other state-of-the-art optimizations
- Increased C++ compile speed
- Windows and DOS Integrated Development Environments
- Visual ObjectBrowser™ to view class relationships at a glance
- DPMI support for compiler and IDE environments gives you huge capacity
- EasyWin™ library makes it easy to convert your DOS programs to Windows
- Resource Workshop for creating Windows user interfaces visually
- Extensive Microsoft® C compatibility
- WinSight™ message tracking utility
- Turbo Debugger® for DOS and Windows
- Turbo Profiler™ for DOS and Windows
- Object-oriented Turbo Assembler®

And with new Borland C++ & Application Frameworks 3.0 you get all of this, plus:

- ObjectWindows™—the application framework for Windows
- Turbo Vision™—the application framework for DOS
- Source code for runtime library and application frameworks

Optimized for professionals

Borland C++ 3.0 ($\$495^{00}$*) or Borland C++ & Application Frameworks 3.0 ($\$749^{00}$*) are optimized for your life-style. But don't wait. Because when it comes to professional programming, there's no better way to earn a living than with Borland C++.

See your dealer today or call now 1-800-331-0877, Dept. 5305

BORLAND

The Leader in Object-Oriented Programming

CODE: 5309

✧ Request 465 on Reader Service Card ✧

Returns

The *div* function returns a structure of type *div_t*, comprising both the quotient and the remainder. The structure shall contain the following members, in either order:

```
int quot;    /* quotient */
int rem;     /* remainder */
```

7.10.6.3 The *labs* function

Synopsis

```
#include <stdlib.h>
long int labs(long int j);
```

Description

The *labs* function is similar to the *abs* function, except that the argument and the returned value each have type *long int*.

7.10.6.4 The *ldiv* function

Synopsis

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

Description

The *ldiv* function is similar to the *div* function, except that the arguments and the members of the returned structure (which has type *ldiv_t*) all have type *long int*.

Using the Integer Arithmetic Functions

Here is a brief summary of the four arithmetic functions:

abs — Call *abs(x)* instead of writing the idiom `x < 0 ? -x : x`. A growing number of Standard C translators generate inline code for *abs* that is smaller and faster than the idiom. In addition, you avoid the occasional surprise when you inadvertently evaluate twice an expression with side effects. Note that on a two's-complement machine, *abs* can generate an overflow.

div — You call *div* for one of two reasons:

- *div* always computes a quotient that truncates toward zero, along with the corresponding remainder, regardless of how the operators / and % behave in a given implementation. This can be important when one of the operands is negative. The expression *(-3)/2* can yield either –2 or –1, while *div(-3, 2).quot* always yields –1. Similarly, *(-3)%2* can yield either 1 or –1, while *div(-3, 2).rem* always yields –1.
- *div* computes both the quotient and remainder at the same time. That can be handy when you need both results. It might even be more efficient if the function expands to inline code that contains only a single divide.

Note that the members of the resulting structure type *div_t* can occur in either order. Don't make any assumptions about the representation of this structure.

labs — is the *long* version of *abs*.

ldiv — is the *long* version of *div*.

Implementing the Arithmetic Functions

Listing 1 shows the file *abs.c*. The absolute value function *abs* is the simplest of the integer math functions. You cannot provide a masking macro, however, because you have to access the value of the argument twice. Some computer architectures have special instructions for computing the absolute value.

That makes *abs* a prime candidate for special treatment as a built-in function generating inline code.

Listing 2 shows the file `div.c`. It provides a portable implementation of the `div` function. You can eliminate the test if you know that negative quotients truncate toward zero. Most computer architectures have a divide instruction that develops both quotient and remainder at the same time. Those that develop proper negative quotients are also candidates for built-in functions. An implementation is at liberty to reorder the members of the structure type *div_t* to match what the hardware generates.

Listing 3 shows the file `labs.c` and Listing 4 shows the file `ldiv.c`. Both define functions that are simply *long* versions of *abs* and *div*.

The C Standard on the Algorithmic Functions

7.10.2 Pseudo-random sequence generation functions

7.10.2.1 The *rand* function

Synopsis

```
#include <stdlib.h>
int rand(void);
```

Description

The *rand* function computes a sequence of pseudo-random integers in the range 0 to *RAND_MAX*.

The implementation shall behave as if no library function calls the *rand* function.

Returns

The *rand* function returns a pseudo-random integer.

Environmental Limit

The value of the *RAND_MAX* macro shall be at least 32767.

7.10.2.2 The *srand* function

Synopsis

```
#include <stdlib.h>
void srand(unsigned int seed);
```

Description

The *srand* function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to *rand*. If *srand* is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If *rand* is called before any calls to *srand* have been made, the same sequence shall be generated as when *srand* is first called with a seed value of 1.

The implementation shall behave as if no library function calls the *srand* function.

Returns

The *srand* function returns no value.

Example

The following functions define a portable implementation of *rand* and *srand*.

```
static unsigned long int next = 1;
   int rand(void)    /* RAND_MAX assumed to be 32767 */
{
   next = next * 1103515245 + 12345;
                  return (unsigned int)
                              next/65536) % 32768;
}
   void srand(unsigned int seed)
{
   next = seed;
}
```

7.10.5 Searching and sorting utilities

7.10.5.1 The *bsearch* function

Synopsis

```
#include <stdlib.h>
void *bsearch(const void *key,
   const void *base, size_t nmemb,
   size_t size, int (*compar
   (const void *, const void *));
```

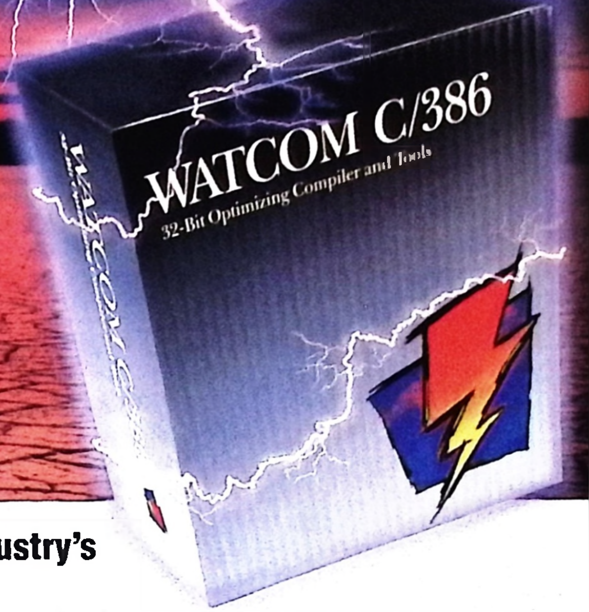
Description

The *bsearch* function searches an array of *nmemb* objects, the initial element of which is pointed to by *base*, for an element that matches the object pointed to by *key*. The size of each element of the array is specified by *size*.

The comparison function pointed to by *compar* is called with two arguments that point to the *key* object and to an array element, in that order. The function shall return an integer less than, equal to, or greater than zero if the *key* object is considered, respectively, to be

Delivering the Power: WATCOM C9.0/386

- **The Widest Range of 32-bit Intel x86 Platforms**
  32-bit DOS, 32-bit Windows, OS/2 2.0, AutoCAD ADS
- **The Industry's Leading Code Optimizer**
  Advanced global optimizer with new 486 optimizations
- **The Most Comprehensive Toolset**
  Debugger, profiler, protected-mode compiler and linker, 32-bit DOS extender with royalty-free run-time, licensed components from Microsoft SDK, and more
- **The Best Value in 32-Bit Tools: $895***



Unleash 32-bit Power!

WATCOM C9.0/386 lets you exploit the two key 32-bit performance benefits. The 32-bit flat memory model simplifies memory management and lets applications address beyond the 640K limit. Powerful 32-bit instruction processing delivers a significant speed advantage: typically at least a 2x speedup.

You Get:

- **100% ANSI and SAA compatible:** C9.0/386 passes all Plum Hall Validation Suite tests
- **Extensive Microsoft compatibility** simplifies porting of 16-bit code
- **Royalty-free** run-time for 32-bit DOS, Windows and OS/2 apps
- **Comprehensive toolset** includes debugger, linker, profiler and more
- **DOS extender** support for Rational, Phar Lap and Ergo
- **Run-time compatible with WATCOM FORTRAN 77/386**

32-bit DOS support includes the DOS/4GW 32-bit DOS extender by Rational Systems with royalty-free runtime license

- Virtual Memory support up to 32Mb

32-bit Windows support enables development and debugging of true 32-bit GUI applications and DLL's.

- Includes licensed Microsoft SDK components

32-bit OS/2 2.0 support includes development for multiple target environments including OS/2 2.0, 32-bit DOS and 32-bit Windows

- Access to full OS/2 2.0 API including Presentation Manager
- Integrated with IBM Workframe/2 Environment

AutoCAD ADS and **ADI** Development: Everything you need to develop and debug ADS and ADI applications for AutoCAD Release 11

Novell's *Network C for NLM's SDK* includes C/386

MICROSOFT WINDOWS. Version 3.0 Compatible Product

PC MAGAZINE EDITORS' CHOICE September 13, 1988 Watcom C Version 6.5

INFO WORLD BEST IN ITS CLASS WATCOM C7.0 1989

COMPUTER LANGUAGE PRODUCTIVITY AWARD 1990

COMPUTER LANGUAGE PRODUCT EXCELLENCE AWARD 1990 JOLT COLA

The Industry's Choice.

Autodesk, *Robert Wenig, Manager, AutoCAD for Windows:* "At Autodesk, we're using WATCOM C/386 in the development of strategic new products since it gives us a competitive edge through early access to new technologies. We also highly recommend WATCOM C/386 to third party AutoCAD add-on (ADS and ADI) developers."

Fox Software, *David Fulton, President:* "FoxPro 2.0 itself is written in WATCOM C, and takes advantage of its many superior features. Optimizing for either speed or compactness is not uncommon, but to accomplish both was quite remarkable."

GO, *Robert Carr, Vice President of Software:* "After looking at the 32-bit Intel 80x86 tools available in the industry, WATCOM C was the best choice. Key factors in our decision were performance, functionality, reliability and technical support."

IBM, *John Soyring, Director of OS/2 Software Developer Programs:* "IBM and WATCOM are working together closely to integrate these compilers with the OS/2 2.0 Programmer's Workbench."

Lotus, *David Reed, Chief Scientist and Vice President, Pen-Based Applications:* "In new product development we're working with WATCOM C because of superior code optimization, responsive support, and timely delivery of technologies important to us like p-code and support for GO Corp's. PenPoint."

Novell, *Nancy Woodward, V.P. and G.M., Development Products:* "We searched the industry for the best 386 C compiler technology to incorporate with our developer toolkits. Our choice was WATCOM."

WATCOM **1-800-265-4555**

The Leader in 32-bit Development Tools

415 Phillip Street, Waterloo, Ontario, Canada. Telephone: (519) 886-3700, Fax: (519) 747-4971. *Price does not include freight and taxes where applicable. Authorized dealers may sell for less. WATCOM C and Lightning Device are trademarks of WATCOM Systems Inc. DOS/4G and DOS/16M are trademarks of Rational Systems Inc. Other trademarks are the properties of their respective owners.

✧ Request 126 on Reader Service Card ✧

less than, to match, or to be greater than the array element. The array shall consist of: all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the *key* object, in that order.[129] [FN129. In practice, the entire array is sorted according to the comparison function.]

Returns

The *bsearch* function returns a pointer to a matching element of the array, or a null pointer if no match is found. If two elements compare as equal, which element is matched is unspecified.

7.10.5.2 The *qsort* function

Synopsis

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
    int (*compar)(const void *, const void *));
```

Description

The *qsort* function sorts an array of *nmemb* objects, the initial element of which is pointed to by *base*. The size of each object is specified by *size*.

The contents of the array are sorted into ascending order according to a comparison function pointed to by *compar*, which is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two elements compare as equal, their order in the sorted array is unspecified.

Returns

The *qsort* function returns no value.

Using the Algorithmic Functions

Here is a brief summary of the four algorithmtic functions:

bsearch — Use this function to search any array whose elements are ordered by pairwise comparisons. You define the ordering with a comparison function that you provide. For example, you can build a keyword lookup function from the basic form as shown in Listing 5.

A few caveats:

- If a key compares equal to two or more elements, *bsearch* can return a pointer to any of these elements.
- Beware of changes in how elements sort when the execution character set changes — call *qsort*, described below, with a compatible comparison function to ensure that an array is properly ordered.
- Be careful using the functions *strcmp* or *strcoll*, declared in *<string.h>*, directly. Both require that strings be stored in the array to be searched. You cannot use them to search an array of pointers to strings. To use *strcmp*, for example, you must write a function pointer argument that looks like *(int (*)(const void *, const void *))&strcmp*.

qsort — Use this function to sort any array whose elements are ordered by pairwise comparisons. You define the ordering with a comparison function that you provide. The comparison function

has a specification similar to that for the function *bsearch*, described above. Note, however, that the *bsearch* comparison function compares a key to an array element. The *sort* comparison function compares two array elements.

A few caveats:

- Don't assume that the function uses the "Quicksort" algorithm, despite the name. It may not. If two or more elements compare equal, *qsort* can leave these elements in any relative order. Hence, *qsort* is not a *stable* sort.
- Beware of changes in how elements sort when the execution character set changes.
- Be careful using the functions *strcmp* or *strcoll* declared in *<string.h>*, directly. Both require that strings be stored in the array to be sorted. You cannot use them to sort an array of pointers to strings. To use *strcmp*, for example, you must write a function pointer argument that looks like *(int (*)(const void *, const void *))&strcmp*.

rand — Call *rand* to obtain the next value in a pseudo-random sequence. You get exactly the same sequence following each call to *srand* with a given argument value. That is often desirable behavior, particularly when you are debugging a program. If you want less predictable behavior, call *clock* or *time*, declared in *<time.h>* to obtain an argument for *srand*. The behavior of *rand* can vary among implementations. If you want exactly the same pseudo-random sequence at all times, copy the version presented here.

Use *RAND_MAX* to scale values returned from *rand*. For example, if you want random numbers of type *float* distributed over the interval [0.0, 1.0], write the expression *(float)rand()/RAND_MAX*. The value of *RAND_MAX* is at least 32,767.

srand — See *rand* above. The program effectively calls *srand(1)* at program startup.

Implementing the Algorithmic Functions

Listing 6 shows the file *qsort.c*. It defines the related function *qsort* that sorts an array beginning at *base*. I introduced the type *_Cmpfun* just to simplify the declaration of arguments for the functions *bsearch* and *qsort*. Don't use this declaration in code that you write if you want it to be portable to other implementations.

This logic is much less simple and more debatable. It is based on the Quicksort algorithm first developed by C.A.R. Hoare. That requires you to pick a partition element, then partially sort the array about this partition. You can then sort each of the two partitions by recursive application of the same technique. The algorithm can sort quite rapidly. It can also sort very slowly.

How best to choose the pivot element is the debatable issue. Pick the first element and an array already in sort eats a lot of time. Pick the last element and an array in reverse sort eats a lot of time. Work too hard at picking an element and all arrays eat a lot of time. I chose simply to pick the last element. That favors arrays that need little rearranging. You may have reason to choose another approach.

qsort calls itself to sort the smaller of the two partitions. It loops internally to sort the larger of the two. That minimizes demands on dynamic storage. At worst, each recursive call must sort an array half as big as the earlier call. To sort *N* elements requires recursion no deeper than $log_2(N)$ calls. (You can sort 1,000,000 elements with at most 20 recursive calls.)

Listing 7 shows the file *bsearch.c*. The function *bsearch* performs a binary search on the sorted array beginning at *base*. The logic is simple but easy to get wrong.

Listing 5

```
#include <stdlib.h>
#include <string.h>

typedef enum {FLOAT, INTEGER} Code;
typedef struct {
    char *s;
    Code code;
    } Entry;
Entry symtab[] = {
    {"float", FLOAT},
    {"integer", INTEGER}}

static int cmp(const void *ck, const void *ce)
    {   /* compare key to table element */
    return (strcmp((const char *)ck, ((Entry *)ce)->s));
    }

Entry *lookup(char *key)
    {   /* lookup key in table */
    return (bsearch(key, symtab,
        sizeof symtab / sizeof symtab[0],
        sizeof symtab[0], &cmp));
    }
/* End of File */
```

Listing 6 (qsort.c)

```
/* qsort function */
#include <stdlib.h>
#include <string.h>

        /* macros */
#define MAX_BUF 256     /* chunk to copy on swap */

void (qsort)(void *base, size_t n, size_t size, _Cmpfun *cmp)
    {   /* sort (char base[size])[n] using quicksort */
    while (1 < n)
        {   /* worth sorting */
        size_t i = 0;
        size_t j = n - 1;
        char *qi = (char *)base;
        char *qj = qi + size * j;
        char *qp = qj;

        while (i < j)
            {   /* partition about pivot */
            while (i < j && (*cmp)(qi, qp) <= 0)
                ++i, qi += size;
            while (i < j && (*cmp)(qp, qj) <= 0)
                --j, qj -= size;
            if (i < j)
                {   /* swap elements i and j */
                char buf[MAX_BUF];
                char *q1 = qi;
                char *q2 = qj;
                size_t m, ms;

                for (ms = size; 0 < ms;
                    ms -= m, q1 += m, q2 -= m)
                    {   /* swap as many as possible */
                    m = ms < sizeof (buf) ? ms : sizeof (buf);
                    memcpy(buf, q1, m);
                    memcpy(q1, q2, m);
                    memcpy(q2, buf, m);
                    }
                ++i, qi += size;
                }
            }
        if (qi != qp)
            {   /* swap elements i and pivot */
            char buf[MAX_BUF];
            char *q1 = qi;
            char *q2 = qp;
            size_t m, ms;

            for (ms = size; 0 < ms; ms -= m, q1 += m, q2 -= m)
                {   /* swap as many as possible */
                m = ms < sizeof (buf) ? ms : sizeof (buf);
                memcpy(buf, q1, m);
                memcpy(q1, q2, m);
                memcpy(q2, buf, m);
                }
            }
        j = n - i - 1, qi += size;
        if (j < i)
            {   /* recurse on smaller partition */
            if (1 < j)
                qsort(qi, j, size, cmp);
            n = i;
            }
        else
            {   /* lower partition is smaller */
            if (1 < i)
                qsort(base, i, size, cmp);
            base = qi;
            n = j;
            }
        }
    }

/* End of File */
```

Notice

WE'VE BEEN ASKED TO MAKE IT VERY CLEAR THAT
THERE'S A WHALE OF A DIFFERENCE BETWEEN QNX® AND UNIX®

WE'RE GLAD TO OBLIGE.

EVEN THOUGH THE QNX REALTIME OPERATING SYSTEM IS *NOT* BASED ON UNIX SOURCE CODE, WE CAN UNDERSTAND HOW SOME DEVELOPERS MIGHT CONFUSE THE TWO. AFTER ALL, QNX FOLLOWS THE LATEST IEEE POSIX 1003.1 AND 1003.2 OPEN SYSTEMS STANDARDS, SO YOU GET THE SAME API AND UTILITY SET FOUND IN MANY UNIX SYSTEMS.

BUT LOOK BENEATH THE SURFACE AND YOU'LL SEE TWO FUNDAMENTALLY DISTINCT ARCHITECTURES. A MONOLITHIC OS ON THE ONE HAND, A MICROKERNEL OS ON THE OTHER. THEY'RE DIFFERENT SPECIES ALTOGETHER, AS DIFFERENT AS A WHALE AND A SCHOOL OF DOLPHINS.

ARCHITECTURE YOU CAN BUILD ON

QNX'S MODULAR, MICROKERNEL ARCHITECTURE GIVES YOU REMARKABLE FLEXIBILITY. OS SERVICES ARE PROVIDED BY A TEAM OF COOPERATING PROCESSES RATHER THAN BY A MASSIVE MONOLITHIC KERNEL. THE RESULT? YOU CAN STRIP QNX DOWN TO A ROM-ABLE EMBEDDED SYSTEM. OR BUILD IT UP TO A VAST NETWORK AND HARNESS THE POWER OF HUNDREDS OF CPUS.

OUR PRICING IS MODULAR TOO, SO YOU'RE NOT FORCED TO PAY FOR WHAT YOU DON'T NEED.

REALTIME PERFORMANCE YOU CAN COUNT ON

NEED THE SPEED OF A FAST, REALTIME EXECUTIVE? QNX CLOCKS IN AT 16 µSEC PER CONTEXT SWITCH ON A 33 MHZ 80486. BUT QNX DOESN'T STOP THERE. WITH ITS PRIORITY-DRIVEN, PREEMPTIVE SCHEDULING, YOU CAN BUILD *REAL* REALTIME SOLUTIONS.

DISTRIBUTED PROCESSING YOU CAN BET ON

QNX CAN TRANSFORM A BUNCH OF ISOLATED MACHINES INTO A SEAMLESS SUPERCOMPUTER, ORCHESTRATING HUNDREDS OF CPUS WITH ITS NETWORK-WIDE IPC. YOU'RE IN COMPLETE CONTROL OF ALL RESOURCES AT ANY POINT, FROM THE PLANT FLOOR RIGHT UP TO YOUR DESKTOP.

SUPPORT YOU CAN DEPEND ON

DURING THE LAST TEN YEARS, WE'VE EARNED A REPUTATION FOR OUTSTANDING TECHNICAL SUPPORT. WE OFFER EVERYTHING FROM 24-HOUR ONLINE CONFERENCING TO ONSITE CONSULTING, SO YOU CAN EASILY REACH THE PEOPLE WHO MAKE UP THE QNX DEVELOPMENT TEAM. THEIR EXPERTISE CAN HELP YOU KEEP YOUR DEVELOPMENT PROJECTS RIGHT ON COURSE.

TO FIND OUT HOW YOUR APPLICATIONS CAN THRIVE IN THE QNX ENVIRONMENT, CALL 1-800-363-9001, (EXT. 102).

QUANTUM SOFTWARE SYSTEMS LTD.
175 TERRENCE MATTHEWS CRESCENT
KANATA, ONTARIO, CANADA
K2M 1W8
TEL: 613-591-0931
FAX: 613-591-3579

QNX®

REALTIME OPERATING SYSTEM

ARCHITECTURE MAKES THE DIFFERENCE

❖ Request 478 on Reader Service Card ❖

Listing 7 (bsearch.c)

```
/* bsearch function */
#include <stdlib.h>

void *(bsearch)(const void *key, const void *base,
    size_t nelem, size_t size, _Cmpfun *cmp)
    {   /* search sorted table by binary chop */
    const char *p;
    size_t n;

    for (p = (const char *)base, n = nelem; 0 < n; )
        {   /* check midpoint of whatever is left */
        const size_t pivot = n > 1;
        const char *const q = p + size * pivot;
        const int val = (*cmp)(key, q);

        if (val  0)
            n = pivot; /* search below pivot */
        else if (val == 0)
            return ((void *)q);  /* found */
        else
            {  /* search above pivot */
            p = q + size;
            n -= pivot + 1;
            }
        }
    return (NULL); /* no match */
    }

/* End of File */
```


Listing 8 (rand.c)

```
/* rand function */
#include <stdlib.h>

/* the seed */
unsigned long _Randseed = 1;

int (rand)(void)
    {   /* compute pseudo-random value */
    _Randseed = _Randseed * 1103515245 + 12345;
    return ((unsigned int)(_Randseed >> 16) & RAND_MAX);
    }

/* End of File */
```

Listing 9 (srand.c)

```
/* srand function */ #include <stdlib.h>

void (srand)(unsigned int seed)
    {   /* alter the seed */
    _Randseed = seed;
    }

/* End of File */
```

Listing 8 shows the file *rand.c*. The function *rand* generates a pseudo-random sequence using the algorithm suggested in the C Standard. That has reasonable properties, plus the advantage of being widely used. One virtue of a random number generator is randomness. Another virtue, ironically, is reproducibility. You often need to check that a calculation based on pseudo-random numbers does what you expect. The arithmetic is performed using *unsigned long* integers to avoid overflows.

Listing 9 shows the file *srand.c*. The function *srand* simply sets *_Randseed*, the seed for the pseudo-random sequence generated by *rand*. I provide a masking macro for *srand*. Hence, the header *<stdlib.h>* declares *_Randseed*, defined in *rand.c*.

References

Donald Knuth, *The Art of Computer Programming*, Vols. 1-3 (Reading, Mass.: Addison-Wesley, 1967 and later). Here is a rich source of algorithms, complete with analysis and tutorial introductions. Volume 1 is *Fundamental Algorithms*, volume 2 is *Seminumerical Algorithms*, and volume 3 is *Sorting and Searching*. Some are in second edition.

You will find oodles of information on:

- maintaining a heap
- computing random numbers
- searching ordered sequences
- sorting
- converting between different numeric bases

Before you tinker with the code presented here, see what Knuth has to say. □

This article is excerpted in part from P.J. Plauger, The Standard C Library, *(Englewood Cliffs, N.J.: Prentice-Hall, 1992).*

Multiple Copy Math Functions

Dr. Timothy Prince

Pipelined architectures including vector and superscalar obtain their speed by depending in part on working on independent calculations in pipeline fashion. Most computationally intensive applications present enough opportunities for parallel or pipeline operation to make worthwhile increases in speed. The normal use of scalar math functions, which produce a single result, poses an obstacle to superscalar performance. These functions can be organized to increase the internal opportunities for parallelism as compared with optimum scalar processor code. Performance remains far short of the potential, unless more parallelism is exploited by working on more than one math function result at a time. A further reason for obtaining multiple results is that the overhead for calling functions which take less than 10 microseconds becomes excessive.

Soon after the introduction of vector computers, vector math functions were introduced to provide vector performance in calculations involving such functions. With pipelined or superscalar processors, vector functions may be effective, but functions which calculate a small number of copies per call are more versatile. Since a typical RISC architecture employs a four-stage pipeline, functions which calculate two or four copies should be enough to maximize performance.

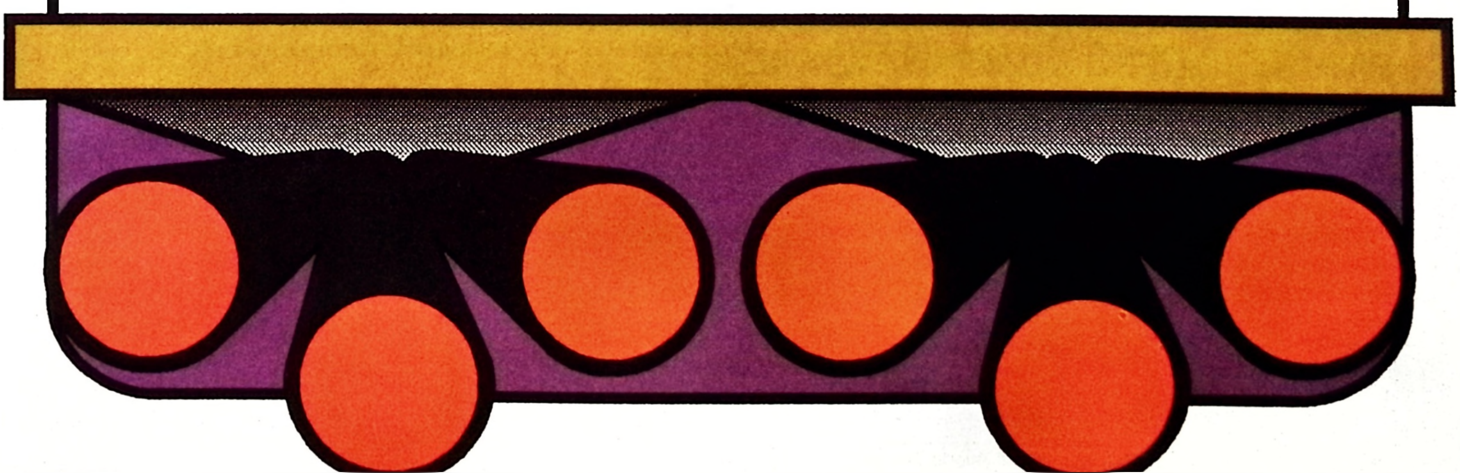
Vector chunk math functions may be used whether or not your compiler makes specific provision for them. I will show actual examples of the C code of such functions. The functions *sin_4* (four *sin*s), *cosf_sinf_2* (two pairs of *float* *sin* and *cos*), *powf_2* (two *float* *pow*s), and *tan_2* (two *tan*s) are chosen for their proven usefulness and because they illustrate the points which I want to make.

Vector vs Superscalar Function Calls

On a vector architecture, vector math functions naturally are performed on argument vectors, and normal vector performance is not approached except on long vectors. These architectures perform well when 50 or more functions are to be calculated at a time. Suppose we wanted to integrate a function involving *sin* and *cos* by Simpson's rule, producing a loop such as

```
for(i=2 ; i<n ; i += 2){
   yint += q[i-2]*sin(t[i-2])
      +4*q[i-1]*sin(t[i-1])+q[i]*sin(t[i]);
   xint += q[i-2]*cos(t[i-2])
      +4*q[i-1]*cos(t[i-1])+q[i]*cos(t[i]);
}
```

Timothy Prince has a B.A. in physics from Harvard and a Ph.D. in mechanical engineering from the University of Cincinnati. He has 25 years of experience in aerodynamic design and computation. He can be contacted at 452 Palmitas St., Solana Beach, CA 92075.



Listing 1 (s)

```
/*
** bc program to calculate Chebyshef economized polynomial
** for evaluation of sin(x) */
/* use bc -l to get c() and s() functions */
define t(x) {                                  /* sin(x)/x */
        if(x==0)return(1.); /* derivative of s function */
    return (s(x) / x); /* put function to be fit here */ }
define b(x) {
    if (x < 0) return (-x);
    return (x); }
define m(x, y) {
    if (x > y) return (x);
    return (y); }
n = 22; /* number of Chebyshef terms */
scale = 40;
p = a(1.) * 4; /* pi */
b = p * .5; /* upper end of curve fit interval */
a = -b; /* lower end of interval */
/* chebft adapted from Press Flannery et al */
/* "Numerical Recipes" FORTRAN version */
for (k = 1; k <= n; ++k) {
c[k] = 0;
f[k] = t(c((k - .5) * p / n) * (b - a) * .5 + (b + a) * .5);
}
/* because of symmetry, even c[] are 0 */
for (j = 1; j <= n; j += 2) {
    s = 0;
    q = (j - 1) * p / n;
    for (k = 1; k <= n; ++k) s += c(q * (k - .5)) * f[k];
    (c[j] = 2 / n * s); }
/* skip even terms, which are 0 */
for (n = 5; n <= 19; n += 2) {
 /* chebpc */
    for (j = 1; j <= n; ++j) d[j] =  e[j] = 0;
    d[1] = c[n];
    for (j = n - 1; j >= 2; --j) {
        for (k = n - j + 1; k >= 2; --k) {
            s = d[k];
            d[k] = d[k - 1] * 2 - e[k];
            e[k] = s; }
        s = d[1];
        d[1] = c[j] - e[1];
        e[1] = s; }
    for (j = n; j >= 2; --j) d[j] = d[j - 1] - e[j];
    d[1] = c[1] * .5 - e[1];
 /* pcshft */
    g = 2 / (b - a);
    for (j = 2; j <= n; ++j) {
        d[j] *= g;
        g *= 2 / (b - a); }
    for (j = 1; j < n; ++j) {
        h = d[n];
        for (k = n - 1; k >= j; --k) {
            h = d[k] - (a + b) * .5 * h;
            d[k] = h; }
    }
    "Chebyshev Sin fit |x|<Pi/2 coefficients"
        " Maximum Rel Error:"
        m(b(c[n + 2]), b(c[2])) / t(b);
    for (i = 1; i <= n; i += 2)  d[i];
}
/* End of File */
```

which involves n evaluations of *sin* and *cos*. A vector compiler would start out by setting up the six vectors made up of the three *sin* and *cos* evaluations from each copy of the loop body. Almost a third of these evaluations would be duplicates, since the vector of *sin(t[i-2])* is the same as the vector of *sin(t[i])* except that *sin(t[0])* and *sin(t[n-1])* are not repeated. Each of these vectors has length $(n+1)/2$, so n would have to be around 100 before good vector performance could be achieved.

In order to approach the performance potential of a vector architecture, we would have to rewrite the code to store the *q[]*sin()* and *q[]*cos()* intermediate results in vectors in a preliminary loop, and then add the appropriate values in another loop. Even after 20 years of vector compiler development, this is more analysis than any compiler can do without human assistance. Most reasonable attempts to improve the performance of this loop for a scalar architecture will prevent vectorization, and changes to improve vector performance will reduce scalar or superscalar performance. Although vector compilers now deal well with sum reductions such as this loop, this is done in effect by splitting the vectors again into six to 10 shorter vectors, making a vector architecture less than fully effective for this type of application.

Unrolling compilers can eliminate most of the duplicate operations by combining the common subexpressions over several iterations of the loop. Each additional loop iteration will require an additional pair of *sins* and *coss*. Compilers have been available (e.g. Multiflow) which would detect this situation automatically and build in a call such as *cosf_sinf_2(t[0],t[1])* which returns

cosf and *sinf* of both arguments, a total of four results from one function call. Such a function is a good match to the architecture of a superscalar processor. Even if your compiler does not perform the transformation automatically, manual rewriting is not unduly burdensome and need not detract from performance on scalar processors. A change as simple as

```
ty = q[0]*sinf(t[0]);
tx = q[0]*cosf(t[0]);
for( i=2; i<n; i+=2){
   temp = cosf_sinf_2(t[i-1],t[i]);
   yint += ty+q[i]*temp.sin2+4*q[i-1]*temp.sin1;
   xint += tx+q[i]*temp.cos2+4*q[i-1]*temp.cos1;
   tx = q[i]*temp.cos2;
   ty = q[i]*temp.sin2;
}
```

should produce most of the advertised performance of any non-vector machine.

Vector and Vector Chunk Function Coding Style

Multiple copy or vector chunk math functions, like vector code, need to be written without conditionals which actually cause transfer of control. In a scalar trig function, it would usually be worth while to test the argument to find out whether it needs to be translated into the primary range. In a vector chunk function, the full range reduction should be performed whether it is needed or not, so that all of the code for the function can be compiled as one basic block.

Transfer of control (branching) gives the compiler a choice of undesirable consequences. Either the pipelines must be allowed to empty, reducing the performance to scalar levels until they are refilled, or *trace scheduling* must be used to fill the pipeline with future operations along the preferred path of execution. Each branch can cause generation of another trace, and the length of compiled code may grow exponentially with the number of branches. The speed gained by keeping the pipelines full may be canceled by increased paging.

A great deal of progress has been made in architecture and compilers in recent years, so that many simple conditional selections can be performed without a transfer of control, if this is necessary to keep a processor producing results at rated speed. This requires calculation of both alternative results followed by instructions which select the correct one. There is a good correspondence with the syntax of *?:* in C, although the compiler should not be totally dependent on the programmer choosing to use *?*. Existing compilers do not vectorize *if..else*. All of the operations in the *sin*, *cos*, *tan*, *exp*, and *log* functions can and should be written in vectorizable form, even when the overall scheme is to favor superscalar execution.

Vector chunk functions do not fit well with the *<errno.h>* system for error reporting. The best that can be done is to report that *ERANGE* or *EDOM* exceptions have been raised for one of the arguments or results. Vector functions give an even hazier indication of trouble.

Some Nuts and Bolts of Machine Dependence

In some of the examples, the sign of a *float* or *double* is tested by assuming that it is in the same position as the sign of an *int* which shares the same storage. This is done either because it is faster or because it reduces register thrashing on certain machines. Generally, in these functions, there is an imbalance of *double* over integer arithmetic, and integer operations can be treated as a free resource whenever *float* operations are being performed at the same time.

This code will work as is on most modern architectures which use the same byte order for *double* and *int*. On VAX-compatible architectures, the sign of a *double* falls in place with the sign of a *short* at the same address, apparently as a result of the PDP-11 ancestry.

A few architectures also suffer from excess of divide and multiply operations over add and subtract, so, in the examples, addition is used instead of multiply by 2. In these examples, it occurs when there are plenty of pipeline slots open, but in other cases, one would not want to prevent an optimizing compiler from converting multiply by 2 to a *ldexp* operation.

As the conditionals which would be required for architectures not conforming to IEEE P754 standards would clutter up the code, I have simply put in *#error* preprocessor directives, which are ignored by non-ANSI compilers because they are indented.

Since a good pipelining compiler will give priority to finishing up the expressions which are placed first in the code, the later copies of the functions tend to fall behind. This may be aggravated by compilers which give priority to loading constants into registers long before they can be used. The way to compensate is to write the earlier copies so as to minimize use of registers and leave more empty pipeline slots which can be filled up by arithmetic from the later copies.

Listing 2 (sin_4.c)

```
typedef struct {
    double X1, X2, X3, X4;
}      ARG_D_4;               /* vector 4 */
#include "float.h"
ARG_D_4 sin_4(xx1, xx2, xx3, xx4)
    double xx1, xx2, xx3, xx4;
/*  use where cos of same argument not needed
** 16 digits precision, compare to 15 digits in "dtrig.h"
** T C Prince */
{
    union dblfmt {
        double dbl;
        int idbl;
        struct dfmt {           /* IEEE p754 */
            unsigned int sign:1;
            unsigned int ex:11;
        }    fmt;
    }      xil;
    double xr, x2, x4, x8;
#ifdef __STDC__
    long double pi = 3.1415926535897932385, pml;
#include <limits.h>
#else
    register double pi = 3.1415926535897932385, pml;
#define LONG_MIN 0x80000000
#endif
    union dblfmt pm, round;
    ARG_D_4 res;
#define BIAS DBL_MAX_EXP
#include <errno.h>
#ifndef errno
    extern int errno;
#endif
#define ODD(i) ((i)&1)
/* use identity sin(x + n pi) = (-1)^n sin(x)
**  to reduce range to -pi/2 < x < pi/2
**      pml=rint(xil/pi) */
#if FLT_ROUNDS != 1
        #error "rounding mode not nearest; adjust code"
#endif
#if FLT_RADIX !=2 && FLT_RADIX != 10
        #error "code not optimum for accuracy in this RADIX"
#endif
#if DBL_DIG > 16
        #error "more terms needed for full accuracy"
#endif
/* shortcut test of sign, not portable to VAX */
    round.dbl = 1 / LDBL_EPSILON;
    xil.dbl = xx1;
    round.idbl |= xil.idbl & LONG_MIN;
    pml = xx1 / pi + round.dbl;
/* sign reversal may reduce register usage */
    xr = pi * (pml -= round.dbl) - xx1;
/* shortcut test for fabs(pml) > INT_MAX */
    pm.dbl = pml;
    if (pm.fmt.ex > BIAS + 31)
        errno = ERANGE;
/* don't wait to calculate xr**2 until sign is fixed;
** another sign reversal is due if pm.dbl is odd */
    x2 = xr * xr;
/* first sign reversal compensated in coefficient signs;
** conditional sign fixed by testing odd/even
** first two results are obtained by straight Horner
** polynomial evaluation */
    res.X1 = (-.9999999999999999 + x2 * (.1666666666666607
    + x2 * (-.833333333328281e-2 + x2 * (.19841269824875e-3
        + x2 * (-.2755731661057e-5 + x2 * (.25051882036e-7
         + x2 * (-.160481709e-9 + x2 * .7374418e-12))))))
        * (ODD((int) pm.dbl) ? -xr : xr);
/* sin(xi2) */
    round.dbl = 1 / LDBL_EPSILON;
    xil.dbl = xx2;
    round.idbl |= xil.idbl & LONG_MIN;
    pml = xx2 / pi + round.dbl;
    xr = pi * (pml -= round.dbl) - xx2;
    pm.dbl = pml;
    if (pm.fmt.ex > BIAS + 31)
        errno = ERANGE;
    x2 = xr * xr;
    res.X2 = (-.9999999999999999 + x2 * (.1666666666666607
    + x2 * (-.833333333328281e-2 + x2 * (.19841269824875e-3
        + x2 * (-.2755731661057e-5 + x2 * (.25051882036e-7
         + x2 * (-.160481709e-9 + x2 * .7374418e-12))))))
        * (ODD((int) pm.dbl) ? -xr : xr);
/* sin(xi3) */
    round.dbl = 1 / LDBL_EPSILON;
    xil.dbl = xx3;
    round.idbl |= xil.idbl & LONG_MIN;
    pml = xx3 / pi + round.dbl;
    xr = pi * (pml -= round.dbl) - xx3;
    pm.dbl = pml;
    if (pm.fmt.ex > BIAS + 31)
        errno = ERANGE;
    x2 = xr * xr;
    x4 = x2 * x2;
/* split into 2 Horner polynomials to increase
** parallelism after 1st result finishes */
    res.X3 = (-.9999999999999999 + x2 * (.1666666666666607
                               + x2 * (-.833333333328281e-2
                                 + x2 * .19841269824875e-3))
            + (-.2755731661057e-5 + x2 * (.25051882036e-7
                                  + x2 * (-.160481709e-9
                    + x2 * .7374418e-12))) * x4 * x4) *
        (ODD((int) pm.dbl) ? -xr : xr);
/* sin(xi4) */
    round.dbl = 1 / LDBL_EPSILON;
    xil.dbl = xx4;
    round.idbl |= xil.idbl & LONG_MIN;
    pml = xx4 / pi + round.dbl;
    xr = pi * (pml -= round.dbl) - xil.dbl;
/* errno is set to ERANGE if any of the arguments are too
** large for reasonable range reduction */
    pm.dbl = pml;
    if (pm.fmt.ex > BIAS + 31)
        errno = ERANGE;
    x2 = xr * xr;
    x4 = x2 * x2;
    x8 = x4 * x4;
/* multiply by 1 is K&R way to enforce parentheses */
    res.X4 = ((-.9999999999999999 + x2 * (.1666666666666607
                             + x2 * (-.833333333328281e-2
                          + x2 * .19841269824875e-3))) * 1
           + (-.2755731661057e-5 + x2 * (.25051882036e-7
       + x2 * (-.160481709e-9 + x2 * .7374418e-12))) * x8)
        * (ODD((int) pm.dbl) ? -xr : xr);
    return res;
}
/* End of File */
```

The later copies are written for more available parallelism at the expense of register usage, so that, when the calculation of the earlier results has finished, the pipelines can still be kept full nearly to the end of the function. This can lead to somewhat greater round off errors in the later copies, in the *double* functions. In the *float* functions, use of *double* arithmetic eliminates the effect of order of operations on accuracy.

Systems which are unable to perform simple conditional selections without branching may require sign changes to be performed by *xor*ing the sign bit. To avoid branching, *errno* may be left alone or set by

```
errno=ERANGE&(-(relational expresssion))
```

which sets it to zero or to *ERANGE*. This is contrary to the normal requirement that *errno* never be set to zero, but may be a satisfactory compromise.

Calculation of Coefficients

Listing 1 shows a *bc* program for calculation of coefficients for *sin*, as used in *sin_4.c*. Running it with *double* arithmetic in C will produce the same results up through at least 10 digits. Because *bc* uses fixed point arithmetic, it needs extra fractional digits for *sin*, more than are needed for most problems. The same program will work if the *t* function is replaced by *a(x)/x*, with appropriate changes in the interval. The coefficients for log base 2, used in *powf_2.c*, are calculated by having the *t* function evaluate the appropriate Taylor-Maclaurin series. With overnight runs, *bc* can calculate coefficients up to 50 significant digits. These Chebyshef subroutines are adaptations of those given by Press, Flannery et al (1).

Multiple Copy *sin*

Listing 2 shows the four copy *sin* function. The code which performs range reduction, by subtracting off the nearest multiple of *pi*, uses a *rint* function, but takes advantage of the fact that dividing by *pi* does not change the sign. It assumes that addition is performed in the highest available precision, which may be more than *double*. *rint* is not covered by standards, and its result may depend on rounding mode, so it would not take care of portability. Use of *long double* precision in these operations is highly desirable, but of little value unless a true *long double* value of *pi* is available. *long double* should prevent degradation of accuracy for arguments up to *pi*10^(LDBL_DIG-DBL_DIG)*.

The sign of the argument is *or*ed into the rounding constant in order not to tie up as many *double* registers, so that the operations on subsequent copies will not be delayed. This procedure avoids branching on processors which do not have a select operation. Portability at the expense of speed can be obtained using expressions such as

```
pm = (int)(x/pi+(x>0?.5:-.5))
```

or

```
pm = (int)(x/pi-.5+(x>0))
```

since, if *fabs(x/pi)* exceeds *INT_MAX*, there probably aren't more than three digits significance left. Since FORTRAN and Pascal have *round double* to integer syntax, certain processors (e.g. MIPS) have implemented it as a single instruction, which is not used by C compilers.

The integer overflow situation is reported as *errno=ERANGE*, without distinguishing which of the four arguments caused it. Non-portable code for testing the exponent field to identify this situation is used because, on the system where the code was tested, there weren't enough *double* registers to squeeze in any more standard arithmetic without stretching the code out by 30%. There are ways to test whether *pm* is odd without ever casting to *int*, so that range errors are avoided out to *pi/DBL_EPSILON*, but it's not worth the trouble.

Covering the whole interval from *-pi/2* to *pi/2* with a single curve fit avoids conditional branches which are particularly troublesome for vector or vector chunk coding. An eight term Chebyshef-economized polynomial is just sufficient to hold the errors to 1 unit-in-the-last-place with *DBL_MANT_DIG = 53*, in the absence of other approximations.

Listing 3 (sin_4~bt.c)

```
/* Tests sin_4() */
typedef struct {
    double X1, X2, X3, X4;
}       ARG_D_4;                    /* vector 4 */
ARG_D_4 sin_4();
#include <math.h>
main(){
        ARG_D_4 res;
        res=sin_4(-2.,-1.,1.,2.);
        printf(
"\t%.17g\t%.17g\n\t%.17g\t%.17g\n\t%.17g\t%.17g\n\t%.17g\t%.17g\n",
                res.X1,sin(-2.),
                res.X2,sin(-1.),
                res.X3,sin(1.),
                res.X4,sin(2.));
}
/* End of File */
```

Putting the interval end points where the function has zero slope helps prevent round off error from introducing discontinuities.

Horner polynomial evaluations are begun before the sign of the result has been determined, leaving the sign switching to be performed when the compiler finds the necessary pipeline slots. The third and fourth polynomial evaluations will lag well behind the first and second, so the third and fourth Horner polynomials are split in two so that the pipelines can be kept fuller after the earlier polynomial evaluations are complete. This adds two multiplications and one possibly significant round off error in each of the third and fourth results.

The fourth copy differs from the third only in that the code is written with parentheses to force the final additions to occur in the most parallel (but not most accurate) order. The dummy multiply by 1 is needed to force K&R compilers to honor the parentheses, but has no effect in ANSI syntax. Since similar techniques are used to a greater extent in scalar math libraries for superscalar processors, these less accurate results are likely to be closer to the scalar results.

This function should achieve a megaflop rating better than the LINPACK rating on many processors, which is unusual effectiveness for such complex code. One of the ways it could be used would be to combine calculation of unrelated *sin*s and *cos*s, using the relationship

```
#define cos(x) sin(PI/2-(x))
```

as needed. A similar tactic should pay off on vector architectures, in which the various arguments are copied to a temporary vector so that the vector *sin* function can be used.

Effective pipelining of this function appears to require more than 16 *double* registers, along with special efforts to perform as many calculations as possible in *int* registers. Examination of results of an early MIPS compiler showed that it was able to economize on the size of generated code by setting the constants only once. Like many RISC architectures, MIPS has immediate constants available only to initialize registers, not to participate directly in

floating point operations. This may not leave enough registers available for extensive pipelining.

Optimization for reduction of length of generated code prior to scheduling of operations is less well correlated with execution speed on pipelined than on scalar processors. The MIPS software does not report the number of empty pipeline stages. The compiler for the original Multiflow 7/200 compiles this code in 96 major instruction cycles and obtains a superscalar speedup factor of 4. Only six of these instructions are empty, all occurring after the first copy result is complete. *sin_4* on the Multiflow is twice as fast as their library *sin*, giving four results in 12 microseconds. On the Silicon Graphics 4D/25, both *sin_4* and the library *sin* take about four microseconds per result.

Listing 3 shows a test driver to compare the results of *sin_4* with *sin*. While many compilers allow passing a *double* to a function which receives it as a union, other compilers push a union on the stack in a different order from a plain *double*. It is safer to make *sin_4* copy the arguments into its unions. On one of the compilers tested, the generated code is the same either way.

The Chebyshef fit of Listing 1 can be changed to use *sin_4*, after changing from *bc* to C syntax. The order of Chebyshef fit may as well be a multiple of 4. The accuracy of math function approximations, such as the functions discussed in this article, can be tested by fitting Chebyshef polynomials and comparing the coefficients with those obtained by a higher accuracy calculation in the same interval.

Multiple Copy Float *cos* and *sin*

Listing 4 shows a function to calculate *cos* and *sin* of two arguments in *float* precision. Since it uses rational polynomial approximations, there is more built-in opportunity for parallelism than in a Horner polynomial, and two such functions are enough to fill a four stage pipeline at the peak stages. Without prototypes, the only way to pass *float* arguments without widening to *double* is by unions. With prototyping, it would be better to pass *float* arguments and copy them to unions inside the function.

One multiply can be eliminated from the critical path by scaling the arguments to multiples of *pi/2* and adjusting the polynomial coefficients accordingly. The division by 2 of the half-angle formulae is buried in the coefficients, so the range reduction maps the arguments into the range -2 to +2. Adding and subtracting *4/LDBL_EPSILON* produces a number which is rounded to the nearest multiple of 4. As long as promotion to IEEE *double* is used, so that no precautions against underflow are needed, there would be no problem in changing the scaling so that the code could start off

```
tn = x1.flt/2/PI - rint(x1.flt/2/PI)
```

in case that could be calculated more efficiently. The choice of scale was influenced by the desire to maintain accuracy if base 16 arithmetic is used.

Scaling the arguments would produce an additional round off error if the calculations were performed in *float* precision, but *double* is almost mandatory anyway as it prevents degradation of accuracy for arguments up to *2pi*10^(DBL_DIG-FLT_DIG)*. A warning such as storing a value into *errno* could be provided when larger arguments arrive, but this is not clearly a failure meriting the *ERANGE* label unless the argument becomes so large that the *rint* code won't work. Basing the *errno* calculation on values which are calculated anyway minimizes the use of additional registers.

The first rational polynomial is calculated Horner style, and the last attempts to catch up by calculating all terms individually, at the cost of one additional multiplication. The scheme of eliminating one of the coefficients by choice of scale allows the numerator to get a head start so that the final multiplication can be performed without delaying the division. The compiler may have to be forced into performing the first add in the denominator without waiting until the last term has been calculated. Certain compilers insist on converting the repeated divisions into multiplications, which is no problem when the operations have been promoted in precision.

Vector Chunk *float pow*

The *pow* function in C is expected to embody two entirely different types of operation. In order for it to be vectorizable, or to obtain good vector chunk performance enhancement with current compilers, it has to be restricted to the cases of positive base, where it can be replaced in effect by

```
#define pow(x,y) exp(log(x)*y)
```

This could be done with a top level *powf_2* which determines whether both pairs of arguments are of one type, and, if so, invokes an appropriate vector chunk function. The usual test is whether *y1* and *y2* are changed by casting to `int` and back to `float`. It doesn't hurt much to use the *log* treatment anyway, unless *x* is negative. If the argument pairs cannot be processed by the same algorithm, it would have been more efficient not to have tried to treat them as a vector chunk at all.

The function of Listing 4 does not take care of the negative base case, which is OK according to ANSI standards if it is called as the implementation of the FORTRAN real exponentiation operator. I use it in this form in time marching aerodynamics codes, where it gets executed millions of times.

Promotion to `double` is really needed only in the sections involving addition of the integer exponent to the base range *log2* up to the splitting of the *exp2* argument into an integer plus or minus a fraction, and then only when the result is far from 1. The somewhat complicated system for splitting the base into modified *frexp* form works quickly and accurately

Listing 4 (powf_2.c)

```
typedef struct {
  float X1, X2;
} ARG_F_2;                      /* vector 2 */
#include "float.h"
#include <errno.h>
#ifndef errno
extern int errno;
#endif
#define MANTBITS (FLT_MANT_DIG -1)
ARG_F_2 powf_2(xi1, y1, xi2, y2)
union fltfmt {
  float flt;
  int iflt;                     /* VAX: must change all this */
  struct ffmt {
        unsigned int ex:9;
        unsigned int mant:MANTBITS;
  } fmt;
} xi1, xi2, y1, y2;
{
#define max(i,j) ((i)>(j)?(i):(j))
#define min(i,j) ((i)<(j)?(i):(j))
#if FLT_MANT_DIG != 24
#error "use portable frexp() ldexp() */
#endif
#if  FLT_ROUNDS == 1
#if defined(__STDC__)
/* This works on some non-ANSI compilers */
#define ROUND(x) ((x)>=0?( \
  (x)+1/LDBL_EPSILON)-1/LDBL_EPSILON: \
                  ((x)-1/LDBL_EPSILON)+1/LDBL_EPSILON)
#else
#define ROUND(x) ((x)>=0?( \
  (x)+1/LDBL_EPSILON)*1-1/LDBL_EPSILON: \
        ((x)-1/LDBL_EPSILON)*1+1/LDBL_EPSILON)
#endif
#else
#define ROUND(x) ((x)>=0?(int)(x+.5):(int)(x-.5))
#endif
  int mi, mi2, msign;
  double xr, x2, r, r1;
  ARG_F_2 res;
/* Copy 1 */
/* This frexp() operation would be done better after
** promotion to double
** but it's not mandatory unless dealing with gradual
** underflow;
** it would eliminate most cases of 0 and Inf changing
** to finite numbers
** if((xi1.flt=frexp(xi1.flt,&mi))<sqrt(.5)){
  --mi;
  xi1.flt *= 2;
  } */
  mi = ((xi1.iflt & 0x7fffffff) -
        (mi2 = (xi1.fmt.mant < 0x3504f3 ?
                        (2 - FLT_MIN_EXP) << MANTBITS :
                        (1 - FLT_MIN_EXP) << MANTBITS)))
        >> MANTBITS;
  if (xi1.iflt < 0 | xi2.iflt < 0) errno = EDOM;
  xi1.iflt = mi2 | xi1.fmt.mant;
/* Mult by y distributed to increase parallelism */
  r1 = (xr = (xi1.flt - 1) / (xi1.flt + 1)) * y1.flt;
  x2 = xr * xr;
/* Coefficients determined by Chebyshef fitting
** double precision is only really needed from here */
  r = y1.flt * (double) mi + r1 * (2.8853904 +
        x2 * (.5958 * x2 + .961588));
/* Msign = (r -= rint(r)) <0 */
  msign = (r -= r1 = ROUND(r)) < 0;
  r *= 125.0718418 + (x2 = r * r);
  x2 = 360.8810526 + 17.3342336 * x2;
/* Xi1.flt = ldexp((x2+r)/(x2-r),(int)r1) */
  xi1.flt = (x2 + r) / (x2 - r);
/* Preferably do this ldexp() operation in double,
** but it's slower,
** even though msign can be eliminated;
** it would always give Inf rather than NaN
** and would allow use of gradual underflow */
  xi1.iflt += (max(FLT_MIN_EXP - 2 + msign,
                min(FLT_MAX_EXP + msign, (int) r1)) << MANTBITS);
  /* X.fmt.ex+=mi; with limiting to prevent exponent wraparound */
  res.X1 = xi1.flt;
/* Copy 2 */
  mi = ((xi2.iflt & 0x7fffffff) -
        (mi2 = (xi2.fmt.mant < 0x3504f3 ?
                        (2 - FLT_MIN_EXP) << MANTBITS :
                        (1 - FLT_MIN_EXP) << MANTBITS)))
        >> MANTBITS;
  xi2.iflt = mi2 | xi2.fmt.mant;
  r1 = (xr = (xi2.flt - 1) / (xi2.flt + 1)) * y2.flt;
  r1 *= x2 = xr * xr;
  r = y2.flt * ((double) mi + xr * 2.8853904) +
        r1 * (.5958 * x2 + .961588);
  msign = (r -= r1 = ROUND(r)) < 0;
  r *= 125.0718418 + (x2 = r * r);
  x2 = 360.8810526 + 17.3342336 * x2;
  xi2.flt = (x2 + r) / (x2 - r);
  xi2.iflt += (max(FLT_MIN_EXP - 2 + msign,
                min(FLT_MAX_EXP + msign, (int) r1)) << MANTBITS);
  res.X2 = xi2.flt;
  return res;
}
/* End of File */
```

without widening on a system without gradual underflow. On architectures such as VAX which use a different byte order for *float* and *int*, the unions and constants are different.

If gradual underflow is to be supported without widening the precision, it will require special case treatment. To reduce degradation of accuracy if widening is not used for addition of the integer and fraction parts of the *log2* function, *log2* should be split into a power of 2 plus a smaller term. This leads to complicated code which may require branching, thus defeating attempts to gain pipelined performance.

Evaluation of `log2(x2)*y2 is speeded up by grouping the terms in pairs. The calculation log2(x2)*y1` then becomes a bottleneck until the multiplication by *y1* is distributed onto the two groups, one of which consists of the three-term Horner polynomial. Multiplication of *y1* by the integer exponent is performed well before it is needed.

Making such detailed adjustments for a given system is possible only with readable assembly language which displays the final scheduling of the pipelined operations, and is helped greatly by static profiling which gives the effective clock count for each block of generated code. Since we try to write these functions so that there is only one code block, and there are few memory accesses which could introduce bus delays, the speed will not depend on data and there should be no question what effect each change has on speed.

In order to make the *ROUND* macro work the same under K&R syntax as it would in ANSI C, dummy multiplications by 1 are introduced. Otherwise it is a matter of luck whether a K&R compiler will generate the required code, although the left associativity of the + and - operators should produce a preference for left to right evaluation. From an algebraic point of view, *ROUND* would do nothing, and AI techniques could conceivably allow a compiler to know this. The peculiar syntax of K&R which requires such multiplications by 1 makes it semi-obligatory for optimizing compilers to eliminate the redundant operation, unless compiling for an architecture which may generate faster code with alternating multiplication and addition.

If the compiler is unable to generate efficient code for the *max* and *min* macros, it would be better to perform the *ldexp* operations on *double*s and hope that the extra range of *double* will take care of over and underflows.

Multiple Copy *tan*

The *tan_2* function (Listing 5) requires the least non-portable coding for optimum results, but it illustrates optimizations which have not appeared in the functions discussed above.

Range reduction consists simply of subtracting the nearest multiple of *pi*, and there is no advantage in playing games with unions. The comparisons start into the pipeline first and are completed before the divides, which may have been converted to multiplications by the compiler.

The remainder of the calculation consists of evaluation of a rational polynomial. On a machine with a divide which pipelines at the same intervals as the other operations, straightforward Horner evaluation of the numerators and denominators might work as well as anything. On the processor for which this code was tuned, a divide operation delays the pipeline, but does not affect addition. For this reason, the order of operations is set up to push all of the multiplications into the pipeline before the divides, as well as to start the first division as soon as possible.

The terms of the numerator and denominator are grouped so that operations are always ready to start into the pipeline, and to take advantage of architectures which have seperate pipelines for multiplication and addition.

In the denominator of the first copy, the high order terms must be added in order of increasing complexity. This could be done with parentheses with an ANSI compiler. Possibly better accuracy could be obtained by adding the high order term last. Order is not dictated in the low order terms of the first copy, so as to avoid delaying the second copy. In the numerator of the second copy, the final multiplications are distributed so that they may be performed before the final addition, in order to get the multiplications out of the way early, as well as to allow the second calculation to begin to catch up with the first.

Listing 5 (cosf_~bs.c)

```
typedef struct {
    float cos1, sin1, cos2, sin2;
}       ARG_FF_2;                   /* vector 2 pairs */
ARG_FF_2 cosf_sinf_2(x1, x2)
    union {
        float flt;
        int iflt;
    }     x1, x2;
{                                   /* 2 pair single precision
                                       sin/cos function */
#include "float.h"
#if FLT_ROUNDS != 1
        #error "rounding mode not nearest, fix code"
#endif
#include <errno.h>
#ifndef errno
    extern int errno;
#endif
#include <math.h>
#define M_2_PI          0.63661977236758134308
#define T2PI            2*M_2_PI
#define TP2             T2PI*T2PI
#define TP3             TP2*T2PI
#define TP4             TP2*TP2
#define TP5             TP3*TP2
    ARG_FF_2 res;
    double tn, td, r;
/* integer compare with 0 same as float, for IEEE
** since arg comes in int register, this is faster
**
** doing everything in double, we won't lose accuracy
** by converting arg to multiple of PI/2
** this allows range reduction by subtracting an integer
**
** reduce to range +- 2, divide by 2 later
** when it cannot underflow */
    tn = x1.flt * M_2_PI + (td = x1.iflt >= 0 ?
                    4 / LDBL_EPSILON : -4 / LDBL_EPSILON);
    if (fabs(x1.flt * M_2_PI) >= 4 / LDBL_EPSILON |
        fabs(x2.flt * M_2_PI) >= 4 / LDBL_EPSILON)
        errno = ERANGE;
    tn -= td;
    tn = x1.flt * M_2_PI - tn;
    td = tn * tn;
/* divide arg by 2 and rationalize numerator and denominator
** numerator of rational approx for tan(x1/2)
** Horner polynomials 1st time */
    tn *= 886.77348 * TP4 + td * (-99.398954 * TP2 + td);
/* denominator */
    td = 886.77346 * TP5 + td *
        (-394.98971 * TP3 + td * 14.425694 * T2PI);
/* cos, sin half angle formulae, rationalized */
    res.cos1 = (td * td - tn * tn) / (td * td + tn * tn);
    res.sin1 = (tn * td + tn * td) / (td * td + tn * tn);
/* copy 2 */
    tn = x2.flt * M_2_PI + (td = x2.iflt >= 0 ?
                    4 / LDBL_EPSILON : -4 / LDBL_EPSILON);
    tn -= td;
    tn = x2.flt * M_2_PI - tn;
    td = tn * tn;
/* distribute terms to finish polynomials quicker */
    tn *= 886.77348 * TP4 - td * 99.398954 * TP2 + td * td;
    r = 886.77346 * TP5 - td * 394.98971 * TP3;
    td = r + td * td * 14.425694 * T2PI;
    res.cos2 = (td * td - tn * tn) / (td * td + tn * tn);
    res.sin2 = (tn * td + tn * td) / (td * td + tn * tn);
    return res;
}

/* End of File */
```

Listing 6 (tan_2.c)

```
typedef struct {
    double X1, X2;
}       ARG_D_2;                    /* vector 2 */
ARG_D_2
tan_2(xi1, xi2)
    double xi1, xi2;
{
    double x2, x, n1, x4;
    ARG_D_2 res;
#include "float.h"
#if FLT_ROUNDS != 1
        #error "rounding mode not nearest; adjust code"
#endif
#if FLT_RADIX !=2 && FLT_RADIX != 10
        #error "code not optimum for accuracy in this
RADIX"
#endif
#include <errno.h>
#ifndef errno
    extern int errno;
#endif
#include <math.h>
#define M_PI    3.14159265358979323846
    x2 = (n1 = (xi1 > 0 ? 1 / LDBL_EPSILON :
            -1 / LDBL_EPSILON)) + (x = xi1) / M_PI;
    x -= (x2 - n1) * M_PI;
    if (fabs(xi1 / M_PI) >= 1 / LDBL_EPSILON |
        fabs(xi2 / M_PI) >= 1 / LDBL_EPSILON)
        errno = ERANGE;
/* now in 1st or 4th quadrant */
#define c0 33281881.3202530279
    n1 = c0 + (x2 = x * x) * (-15666569.8711211851);
    x4 = x2 * x2;
    res.X1 = x * (c0 + x2 * (-4572609.43103684572) + x4 *
        (131095.887915363619 + x2 * (-968.863245687503149 +
                x2))) / (n1 + x4 * (915701.668921990803
                        + x2 * (-13491.7937027796916)
                        + x4 * 44.4083322286368691));
/* copy 2 */
    x2 = (n1 = (xi2 > 0 ? 1 / LDBL_EPSILON :
            -1 / LDBL_EPSILON)) + (x = xi2) / M_PI;
    x -= (x2 - n1) * M_PI;
    n1 = 915701.668921990803 - (x2 = x * x)
        * 13491.7937027796916;
    x4 = x2 * x2;
    res.X2 = (x * (c0 + x2 * (-4572609.43103684572)) +
        (131095.887915363619 + x2 * (-968.863245687503149 +
        x2)) * x4 * x) / (c0 + x2 * (-15666569.8711211851) +
                x4 * (n1 + x4 * 44.4083322286368691));
    return res;
}
/* End of File */
```

Summary

Much of this article will appeal only to those who like to tweak code for another 20% in performance. I have concentrated on the points where architectural dependencies pop up and tried to show where their impact can be reduced for relatively small performance penalties. Math library functions are probably the closest thing to applications where non-portable code is appropriate. This is the reason for these functions (at least the scalar versions) being defined in the C standard so that they need not be carried as part of on application.

The extent to which special vector chunk functions should be used to perform math library operations in groups may be questioned. An application which uses these functions probably should provide an alternative header file which will cause them to be replaced with standard functions. Compilers are most likely to begin to incorporate such functions automatically if they produce benefits on the standard benchmarks.

Scalar math functions can detract from the performance of superscalar processors. The techniques shown enable superscalar performance to be obtained in the evaluation of grouped math library functions. In typical applications, the percentage of execution time spent in math functions can be reduced in comparison with a scalar processor. ☐

References

John Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, 1985 (avoidance of code explosion, trace scheduled pipelined code).

Press, Flannery, Teukolsky, Vetterling, *Numerical Recipes in C*, Cambridge, 1988 for Chebyshef analysis.

John Palmer, Stephen Morse, *The 8087 Primer*, Wiley, 1984 for some math function concepts.

P. J. Plauger, Jim Brodie, *Standard C*, Microsoft Press, 1989 for the simplest satisfactory explanation of *float.h*, *limits.h*, *math.h*.

T. Prince, "Generating Source for <*float.h*>," *The C Users Journal*, V8N6, June 1990.

Porting Command Line

William Smith

Not very long ago the computer industry's direction concerning operating systems and user interfaces was uncertain and confusing. You had to be psychic, lucky, or just plain good at reading the writing on the wall to assess the direction the industry was going. To name just a few of the possibilities, there was UNIX and X-Window, OS/2 and Presentation Manager, MS-DOS and a primitive Windows 2.0, and MS-DOS and countless lesser known third party user interface libraries. Right at the height of this confusion, I was faced with choosing an operating system and user interface for a software development project. I was developing a data acquisition and data management system for an automated athletic weight training system. The customer also wanted the program to have a modern graphical user interface (GUI).

For economic reasons and availability of development tools, I decided to write the software in C for a target 80386 personal computer running MS-DOS. The user interface decision was much harder. I, like many people when faced with a tough decision, decided not to decide. I made an engineering decision to do the user interface last. This allowed me to postpone the commitment to a specific user interface library. This was contrary to many opinions about designing software at the time. Instead of designing the software from the user interface and screen level first, I designed the software by identifying core functionality and operations independent of the user interface. To get the project going, I specified that the software was to be first developed using a simple command line interface (CLI). Later, when a graphical user interface library was chosen, I would port the program to work with the GUI library. As soon as a clear direction in the software industry emerged, I would decide upon what GUI to use. Much to my delight and the satisfaction of the customer, this approach eventually paid off.

Well as most of you know, Windows 3.0 came out and quickly became a success. It became obvious what user interface to use. Eventually the project was over and the customer ended up with both a command line version and after a successful port, a Windows version of the software. Along the way I learned a lot about the approach involved in making this port easy. I am going to share with you some of what I learned about porting command line interfaces to GUI's in general and Windows in particular.

The concept of porting I am going to discuss requires work. It is not a simple recompile under a different environment. Granted, there are some features built into Microsoft QuickC for Windows

William Smith is the engineering manager at Montana Software, a software development company specializing in custom applications for MS-DOS and Windows. You may contact him by mail at P.O. Box 663, Bozeman, MT 59771-0663.

User Interfaces to GUIs

and version 3.0 of Borland C++ that allow you to recompile your code under Windows with no changes. The approach is easy, but all it gets you is a command line within a window. I do not consider this a true GUI. To get true GUI behavior in your program you are going to have to write some code and make some changes. GUIs are here to stay and will become increasingly popular in the future. The effort to port your code to a GUI is worth it. With proper planning, it does not have to be that painful either.

CLI Versus GUI

With the popularity of GUIs, CLIs may seem antiquated, but they have their place. They are easy to write and if you stick to the standard C library, very portable. You can generate a test program for exercising new code quicker using a CLI then a GUI. CLIs are fast and can be easy to use. All that the program requires of the user is to type in the program name and some options at the operating system prompt. The command line interface is elegant in its simplicity and appreciated by the skilled software user. CLIs are also very adaptable to batch mode processing. Unfortunately, CLIs are cryptic and require the user to have knowledge and memory of the command line syntax. By providing a help screen defining proper usage, you can relieve this challenge somewhat for the inexperienced user. An easy to use CLI program should provide a provision to invoke this help screen when an *h* or *?* option is passed on the command line. The program should also display the help information when there is an error in the command line syntax.

GUIs are visually appealing and easier to negotiate then CLIs especially for the unskilled user. GUIs can require more steps to accomplish the same task then a CLI and are not as conducive to batch mode processing as CLIs. With some effort, you can add key stroke short cuts and batch ability to GUIs to satisfy the demands of the advanced user.

Table 1 contains a list of user interface characteristics and features. It rates CLIs and GUIs for comparison.

CLI Translated to GUI

Programs in the simplest terms require input, perform some task and generate output. The input information is in the form of instructions and data. The output information consists of results and data. With a command line interface your choices of how to communicate instructions to a program are limited. Typically, CLIs use single characters, sometimes preceded by a delimiting character such as /, to specify options, commands, or flags. The user passes data to CLIs in the form of file names or lists of strings. On the other hand there are many more options available on how to communicate information to a program that employs a GUI. Table 2, lists the basic command line interface elements and corresponding GUI elements. The nomenclature is based on Windows. Notice there is not a one-to-one correspondence between a command line element and a GUI element. With a GUI, there can be many different ways to accomplish the same task. This gives the developer some flexibility in designing the interface.

Listing 1 contains a code fragment from the `main` function of a program that processes a command line and performs database operations. This is an excerpt from a data base utility program that I first created as a CLI program and later ported to Windows. The command line is simple. To specify an operation, the program requires a single character as the first argument on the command line. The second and third arguments are a file name and a key name. The operation chosen determines which of these last two arguments are required.

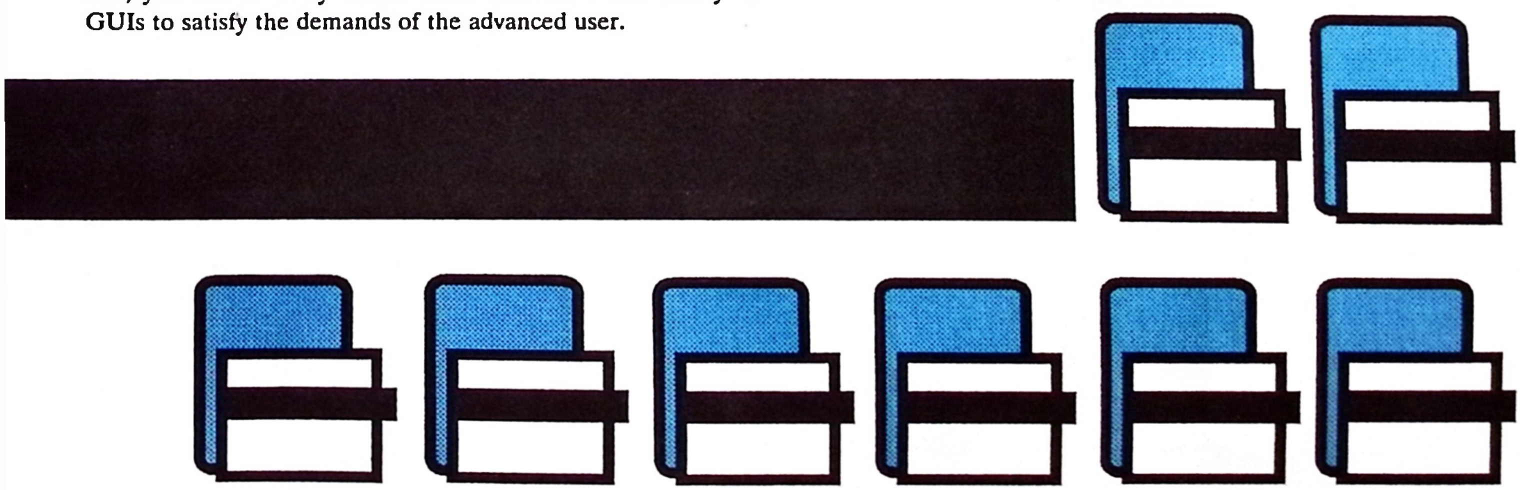


Table 1

Characteristics / Features	CLI	GUI
Ease of Development	Good	Poor
Portability	Good	Poor
Batch Processing	Good	Poor
Ease of Use	Poor	Good
Aesthetics and Presentation	Poor	Good
Visibility of Information	Poor	Good
Flexibility of Program Operation	Poor	Good
Number of Interaction Steps	Few	Many

User Interface Issues, CLI Versus GUI

Table 2

CLI Element	Corresponding GUI Elements
Option / Action Flag	Menu Item, Radio Button, Check Box
String Unlimited Choices	Edit Box, Check Box
String Limited Choices	Menu Item, Radio Button, List Box
File Name	Edit Box, List Box, Dialog Box
Script File	Editor, Dialog Box, Custom Builder

CLI Elements and Analogous GUI Elements

Listing 1

```
switch ( argv[1][0] )
    {
    case 'a':
    case 'A':
        status = add_data_to_db( argv[2] );
        break;
    case 'd':
    case 'D':
        status = del_data_from_db( argv[2] );
        break;
    case 'g':
    case 'G':
        status = get_data_from_db( argv[2], argv[3] );
        break;
    case 'l':
    case 'L':
        status = list_keys_in_db();
        break;
    case 'r':
    case 'R':
        status = replace_data_in_db( argv[2], argv[3] );
        break;
    case 'v':
    case 'V':
        status = vrfy_data_in_db( argv[2], argv[3] );
        break;
    default:
        status = FAIL;
        break;
    }   /* switch ( argv[1][0] ) */

/* End of File */
```

Listing 2

```
switch ( wParam )
    {
    case IDM_ADD:
        /* Select a File */
        status = FileSelectDialog( hInstance, hWnd,
                Caption, FileSpec );
        if ( status == FAIL || status == FALSE )
            break;   /* Canceled the operation. */
        /* Add file to database */
        status = add_data_to_db( FileSpec );
        break;
    case IDM_DELETE:
        /* Select a Key */
        status = KeySelectDialog( hInstance, hWnd,
                Caption, Key );
        if ( status == FAIL || status == FALSE )
            break;   /* Canceled the operation. */
        /* Delete Keyed Data from database */
        status = del_data_from_db( Key );
        break;
    case IDM_GET:
        /* Select a Key */
        KeySelectDialog( hInstance, hWnd,
                Caption, Key );
        if ( status == FAIL || status == FALSE )
            break;   /* Canceled the operation. */
        /* Select a File */
        FileSelectDialog( hInstance, hWnd,
                Caption, FileSpec );
        if ( status == FAIL || status == FALSE )
            break;   /* Canceled the operation. */
        /* Get data from database and store in File */
        status = get_data_from_db( Key, FileSpec );
        break;
    case IDM_REPLACE:
        /* Select a Key */
        KeySelectDialog( hInstance, hWnd,
                Caption, Key );
        if ( status == FAIL || status == FALSE )
            break;   /* Canceled the operation. */
        /* Select a data file */
        FileSelectDialog( hInstance, hWnd,
                Caption, FileSpec );
        if ( status == FAIL || status == FALSE )
            break;   /* Canceled the operation. */
        /* Replace Key data with data in FileSpec */
        status = replace_data_in_db( Key, FileSpec );
        break;
    case IDM_VERIFY:
        /* Select a Key */
        KeySelectDialog( hInstance, hWnd,
                Caption, Key );
        if ( status == FAIL || status == FALSE )
            break;   /* Canceled the operation. */
        /* Select a File */
        FileSelectDialog( hInstance, hWnd,
                Caption, FileSpec );
        if ( status == FAIL || status == FALSE )
            break;   /* Canceled the operation. */
        /* Verify Data in database (Key) matches
        ** data in FileSpec */
        status = vrfy_data_in_db( Key, FileSpec );
        break;
    default:
        status = FAIL;
        break;
    }   /* switch ( wParam ) */

/* End of File */
```

The command line syntax is

```
PROGRAM OPERATION FILE KEY
```

The possible operations are shown in Table 3.

Listing 2 contains an excerpt from a Windows program. This code is taken from a program that accomplishes the same tasks as the program that contains Listing 1 . The code is from the windows procedure for the main window that responds to messages for the main window. The code fragment contains a switch statement that reacts to menu choices that are in the form of messages communicated from Windows. There is correspondence between the command line options and the menu choices. The user specifies the file and key through interaction with dialog boxes. Notice that the calls to the functions, *add_data_to_db*, *del_data_from_db*, *get_data_from_db*, *replace_data_in_db*, *vrfy_data_in_db* are the same for both the CLI version and the GUI version. The code for these function should be portable across user interfaces.

Table 4 lists the CLI arguments and the corresponding GUI elements used to accomplish the same operations.

GUI Portability Strategy

There are three major guidelines that form the foundation of a strategy for portability between user interfaces:

1. Identify high level functionality and data structures
2. Isolate program functionality from user interface code
3. Use standard library and standard types

Planning for user interface portability requires adopting a design philosophy of first identifying high level program functionality and avoiding the specifics of screen design. The idea is to isolate the major data structures and operations that the program supports. If you can wrap a command line interface around the operations that your program performs, you are on the right track. Granted some programs such as word processors do not lend themselves to command line interfaces. Even in this situation, you can isolate individual functions a word processor performs and group them in a utility program with a command line interface.

Table 3

Operation	Description
A	add data contained in FILE to database
D	delete data specified by KEY from database
G	get data specified by KEY from database and store in FILE
L	list all the keys in the database to standard output
R	replace data in database specified by KEY with data in FILE
V	verify data in database specified by KEY with data in FILE

Once you define the major functionality, make sure you strongly segregate the code you write to implement this functionality from the user interface code. The modules that contain the core operations of your program should be portable and not make any function calls to a user interface library.

To help with portability, use the standard library functions and the standard types. Some of the issues encountered when porting among GUIs and operating systems are sizes of standard types, structure packing, and alignment. The size of some of the standard types will change from one platform to another. An example is the default *int* type. Under some compilers an *int* is 16 bits while with others it is 32 bits. If you do not care what size it is and want to use the native most efficient size, use just a plain *int*. If you only need 16 bits and want to conserve space in a platform where *int* is 32 bits use *short int*. If you need 32 bits even in a platform where *int* is 16 bits use *long int*. Avoid *typedef*ing *int* and encoding the size in the type such as *int16* or *int32*. Some claim that this increases portability, but I have found the exact opposite to be true. I also recommend using the *size_t* type defined in standard C. *size_t* is defined as an unsigned integer. It is convenient to use variables of type *size_t* as array indexes.

Table 4

CLI Arguments	GUI Elements
Option	Options (Menu)
A	Add...
D	Delete...
G	Get...
L	
R	Replace...
V	Verify...
File name	File Select Dialog Box
Key name	Key Select Dialog Box

CLI Arguments and Corresponding GUI Elements

Related to type size is structure packing and alignment. In most situations, compilers align structure members (except *chars*) on boundaries that correspond to the most efficient type size. On a 16-bit system, structure members are aligned on word boundaries. On a 32-bit system, structure members are aligned on *double* word boundaries. Some compilers allow the program to control structure alignment.

Try to avoid dependencies in your code on type size and structure alignment. The *sizeof* operator can help with types and the *offsetof* macro can help with structures. Pay careful attention to third party libraries if you use them. They may have size and structure alignment dependencies that could bite you later.

Buffer sizes and string lengths should be set using manifest constants. For example, the maximum length of a string to hold a file name may change from one system to another. It is far easier to change the definition of a manifest constant in one place than find all places where space for a string is allocated.

Porting to Windows — the Gruesome Details

Since Windows is such a popular GUI, it is worth talking about some of the specific issues encountered when porting existing C code to this environment. As a first step in porting your CLI program to the Windows GUI, you may want to create a user interface shell and spawn the CLI version of your program using the Windows *WinExec* function call. *WinExec* is similar to the *spawn* function family in standard C. This approach will get you up and running, but I found it unacceptable for a finished product. The major drawback is the lack of and the difficulty involved in communicating between MS-DOS and Windows programs.

The next step is to replace the CLI interface and compile your code as a Windows application. Unfortunately, even if you prepared ahead for portability there are some problems that may surface.

Types and Structure Alignment

I have already mentioned data types and structure packing. They are especially important issues under Windows. Windows, in its present incarnation, is a 16-bit environment and the default *int* type is 16 bits, but Windows requires structures to be aligned on eight-bit (byte) boundaries. If your code expects structures to be aligned on 16-bit (word) boundaries this may affect you. I ran into alignment problems with a third party database library. The situation forced me into hand padding my structures so members greater than a single byte in size were aligned on word boundaries. I inserted eight-bit padding members of type *char* after an odd number of single byte members.

The Windows programing environment contains many new types defined in the *include* file, *WINDOWS.H*. I recommend you use these types, but confine their usage to the user interface portions of your code.

Memory Models

Since Windows runs under MS-DOS and is subject to the caveats of the Intel segmented architecture, you will have to deal with near and far pointer issues and memory models. Since I wanted as much of my code to be as standard C-like as possible, I tried to avoid sprinkling my code with the keywords *near* and *far* that are not standard C keywords. The general wisdom on Windows claims that programs compiled using the small or medium memory model behave better under Windows than those compiled with the large or compact memory models. Using the small or medium memory model forces you to declare pointers with the *far* keyword if they happen to be in far heap space. Even though using the large and compact memory models is discouraged, many of the Windows library functions also require far pointers as parameters. The only way to get far pointers without adding the *far* keyword to every declaration is to use the large or compact memory model. Windows does not like programs compiled under these memory models because they may contain multiple data segments. Windows fixes multiple data segments in memory. This situation prevents Windows from running more than one instance of such programs and may cause inefficiencies in Windows memory management. This is the case with Microsoft C, but not always with Borland C++. Yes, that is right, the two compilers have a slightly different implementation of the large and compact memory models. Microsoft C creates multiple data segments when using the large or compact memory model and you have little control over the outcome. Borland C++ creates a single data segment unless you specifically tell it to create more. You can run multiple instances of a program compiled under Borland C++ using the large or compact memory model. The exact program compiled with Microsoft C using the large memory model will run as a single instance only.

I have tried the large memory model under Windows and did not notice any performance problems with Windows in standard or enhanced mode. I never even tried real mode. Since Windows 3.1 eliminates real mode, I probably never will use real mode. If you need pointers to far data, your choices are to use the large memory model or to use mixed model programming by declaring pointers with the *far* keyword.

Dynamic Memory

Windows does support the *malloc* family of standard C library memory management functions. Unfortunately, they may have slightly different behavior then what you are use to. Depending on the memory model, *malloc* may allocate memory in the near heap. If you want to force allocation in the far heap independent of memory model, you will have to use the function *_fmalloc*. Since Windows maps *_fmalloc* to the Windows function *GlobalAlloc*, there is a limitation on how many times you can call *_fmalloc*. Every time you call

GlobalAlloc, Windows uses a segment selector. There is a finite number of segment selectors available. This happens to be 8192 for all of Windows – not just your application. If your program requires the allocation of a lot of small pieces of memory, you can quickly run out of selectors even if you still have lots of free memory. The solution to this problem is to call *GlobalAlloc* sparingly and use subsegment allocation. This means you will have to write your own memory manager or buy one of the third party libraries on the market. Version 3.0 of Borland C++ supports subsegment memory allocation and eliminates this problem. I expect eventually all compilers that support Windows will support this feature.

WINSTUB.EXE

Windows allows a non-Windows program to be bound to your Windows program. You specify the stub program in the linker definition file. MS-DOS executes the stub program when you invoke the program from the MS-DOS prompt. I took advantage of this feature to bind the command line or MS-DOS version of a program to the GUI or Windows version of the same program. The only problem I encountered with this was that Borland C++ enforced a 64KB maximum size limitation on the stub program. Microsoft C allowed the stub program to be any size.

UAEs and New Bugs under Windows

When I first compiled my program under Windows as a Windows application, I was extremely disappointed when it would not run without generating UAEs (Unrecoverable Application Errors). Upon tracking down the offending lines of code, a pattern started to emerge. The majority of the UAEs where caused by dereferencing null pointers. This was occurring in the code I had written and also in the standard library code that I was passing null pointers to as parameters. Since the program worked fine under MS-DOS, there was some argument among co-workers about whether these were actual bugs. One of my partners claimed that functions such as *strcmp* should be able to handle a null pointer parameter. Since I could not find any reference that specified how some of the standard library functions responded to null pointers as parameters, I decided to be conservative on this issue and actually modified the program's code to avoid passing null pointers to functions where the behavior was not defined and caused UAEs. I recommend that you be careful about dereferencing null pointers and passing null pointers to standard library functions where the behavior is not specifically defined by the standard or defined in the function description that comes with your compiler's documentation.

Conclusions

Ease in portability between user interfaces requires planning. A decision to first develop an application with a command line interface and then port it to Windows made me deal head on with GUI portability problems. What I eventually ended up with is a program where most of the code will port to any GUI without rewriting it.

Planning for portability requires you to identify the needed operations and the high level data elements independent of any user interface issues. You should isolate user interface specific code from the core program code. You should be able to access the functionality of your program through a simple command line interface. This can be handy for testing. For a CLI, the *main* function should do nothing but process the command line and make the requested function calls. These same function calls can then be called in a similar way when responding to messages or events in a program with a GUI. The GUI program will have to be more than just a simple *main* function module and may require many new modules that support the GUI functionality and screens. The goal is to have the business end of the code that does the data crunching and calculating remain unchanged when porting from one user interface to another. □

A Versatile Menu Program for Turbo C

Roger T. Stevens

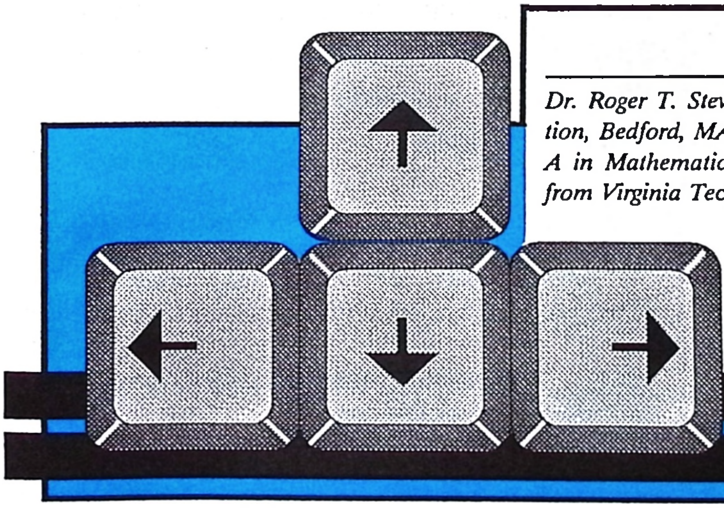
Introduction

Although there are menu programs available commercially and as shareware, most of them provide only the capability to display and select from a list of menu items, and do not permit interaction with the display. However, interaction with the display is often essential for efficient operation of a program. Look at the sample display of Figure 1. The two bottom lines are provided by the menu function. They give the user the choice of the actions: *DISPLAY DATA*, *CHANGE DATA*, *ADD DATA*, or *QUIT*. All you can do with most menu functions is to select one these actions. However, for the first two of these actions, we want to interact with the display, by selecting a name from the list for the action to be performed upon. The menu function described below will automatically switch mode after an action is selected. For the first two actions, the new mode can permit scanning the list of names with the up and down cursor arrows, highlighting each in turn with a contrasting color combination. The user then hits *ESC* to activate the selected function. The cursor is only permitted to go to the first letter of each name in the list.

Now look at the display of Figure 2. This is a new user generated display which provides detailed data on the person selected from the first display. Again the menu function is activated; this time it just displays two lines of instructions at the bottom of the screen. This is the display that you see if you chose the action *CHANGE DATA*, from the first display. Those places on the screen where you are allowed to change the data are actually shown on this display in a different color from the rest of the display and the cursor is restricted to only these positions. After you have modified the data in any way you desire, hitting *ESC* returns to the main program. The user can then arrange to read any data changes from the screen into his data files. Note that as long as you are within the menu function, you have the capability to move the cursor to any permissible location and change or rechange the data there. If at the first screen, you selected the *DISPLAY DATA* action, the bottom explanatory lines simply say Hit any key to continue... and no modification of display data is permitted.

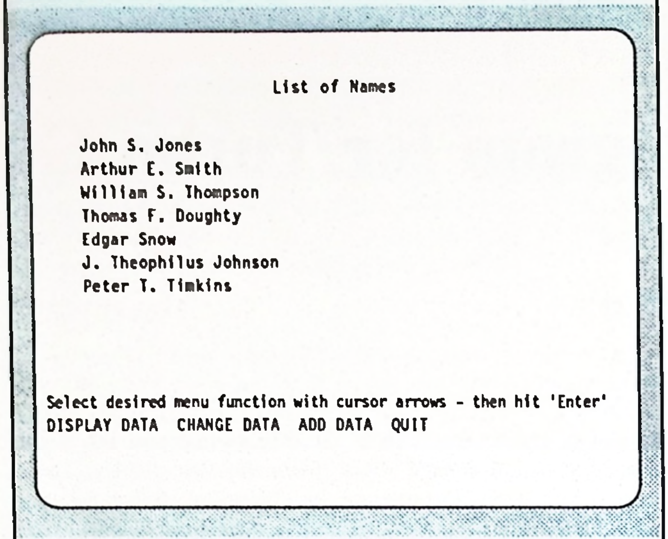
For the *ADD DATA* action, no selection from the list of names is permitted; instead the display immediately switches to that of Figure 2, but with the data areas blank, ready to receive data on a new person.

All of the menu and display interaction is controlled by a versatile menu function which provides a number of different modes of operation through passed parameters. This menu function is described in the text that follows, and the code is in Listing 1. The listing also includes a number of useful functions needed for program operation.



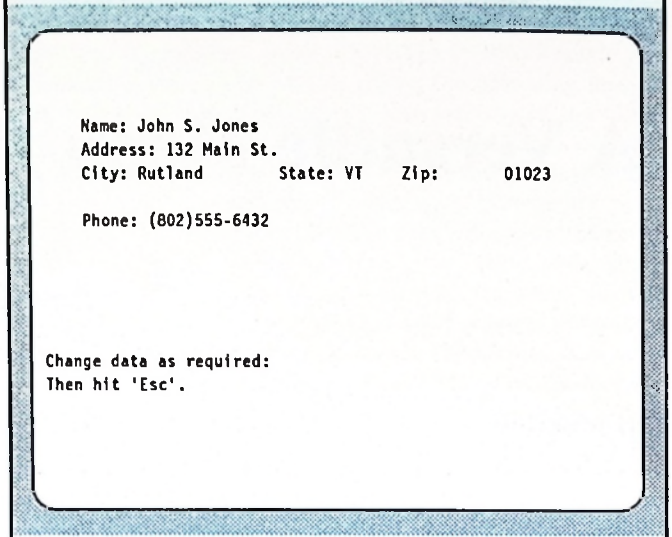
Dr. Roger T. Stevens is a member of the technical staff of the MITRE Corporation, Bedford, MA. He holds a B. A. degree in English from Union College, an M A in Mathematics from Boston University, an M. Eng. in Systems Engineering from Virginia Tech, and a PhD. in Electrical Engineering from California Western University. Dr. Stevens' books Graphics Programming in C, Fractal Programming in C, and Fractal Programming in Turbo Pascal are published by M & T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063.

Figure 1



Name List Display

Figure 2



Display of Detailed Individual Data

What You Can Do with the Menu Program

The menu program begins by displaying two adjacent lines of instructions or menu items on the screen. You can select the location of the first of these lines by setting a parameter called *first_line_loc*, which is passed to the menu function. The first line normally consists of general instructions; the second line, which is the next line after the first line, can be a list of menu items from which the user makes a selection, or can be another set of general instructions, depending upon the mode of operation of the menu program.

Table 1

BACKGROUND	Black	Blue	Green	Cyan	Red	Magenta	Brown	Light Gray
FOREGROUND								
Bright White	15	31	47	63	79	95	111	127
Yellow	14	30	46	62	78	94	110	126
Light Magenta	13	29	45	61	77	93	109	125
Light Red	12	28	44	60	76	92	108	124
Light Cyan	11	27	43	59	75	91	107	123
Light Green	10	26	42	58	74	90	106	122
Light Blue	9	25	41	57	73	89	105	121
Dark Gray	8	24	40	56	72	88	104	120
Light Gray	7	23	39	55	71	87	103	119
Brown	6	22	38	54	70	86	102	118
Magenta	5	21	37	53	69	85	101	117
Red	4	20	36	52	68	84	100	116
Cyan	3	19	35	51	67	83	99	115
Green	2	18	34	50	66	82	98	114
Blue	1	17	33	49	65	81	97	113
Black	0	16	32	48	64	80	96	112

Color Table

Listing 1

```
/*
menu = controls cursor movement and menu display and selection -
       returns the number of the selected menu item.

       values:                a string consisting of two digits showing
                              the number of menu choices, followed by
                              four digits for each choice, the first
                              two showing the starting column of the
                              menu item and the second two its length.

       menu_first_line:       the first line of instructions in the
                              menu mode.

       menu_second_line:      the second line in the menu mode. It con-
                              tains the menu choices.

       screen_first_line:     the first line of instructions in the
                              screen mode.

       screen_second_line:    the second line of instructions in the
                              screen mode.

       menu_type:             0: Start with Menu Mode.
                                 Alphanumeric characters are ignored
                                 when entered in screen mode.
                              1: Start with Menu Mode.
                                 Alphanumeric characters are displayed
                                 when entered in screen mode
                              2: Start with Screen Mode.
                                 Alphanumeric characters are ignored
                                 when entered in screen mode.
                              3: Start with Screen Mode.
                                 Alphanumeric characters are displayed
                                 when entered in screen mode.

       screen_color:          color for screen mode.
       menu_color:            color for menu display.

       highlight_color:       color of selected menu item

       select_no:             number of menu items using selection

       escape_char:           number of the key selected for escape
                              from the screen display.

       map:                   bit map of permitted cursor locations.
                              The cursor will go to permitted locations
                              only and no others.
*/

int menu(char values[],char menu_first_line[],char menu_second_line[],
     char screen_first_line[],char screen_second_line[],int menu_type,
     int screen_color,int menu_color,int highlight_color, int select_no,
     int escape_char, int first_line_loc, char map[25][10])
{
        union REGS reg;
        int i,choices,indx,start,length,menu_second_line_length;
        int interim,remainder,temp;
        char spaces[80],prev_char;

        for (i=0; i<76; i++)
                spaces[i] = ' ';
        spaces[76] = '\0';
        menu_second_line_length = strlen(menu_second_line);
        gotoxy(2,first_line_loc);
        choice = 1;
        if (menu_type <= 1)
        {
                color_printf("%s",menu_color,menu_first_line);
                gotoxy(2,first_line_loc+1);
                length = values[4] - '0';
                length = 10 * length + values[5] - '0';
```

You can specify the color combination for the menu items and the color combination which is used to highlight the currently selected menu item. The rest of the screen display remains as you generated it before calling the menu function. A menu item is selected by use of the cursor arrows; when selection is complete, this phase of the menu program is terminated by hitting the *Enter* key. For any number of menu items, beginning with the leftmost, you can specify an alternate (screen) mode of operation, which is entered after the *Enter* key is hit. You can specify each permissible cursor position for this alternate mode, and the cursor will only be allowed to go to these selected positions. (For example, if the down arrow is hit, the menu function will look at the current column and the next line and move the cursor there if it is a permissible position. If that position is not permissible, the function will look for the nearest permissible position on that line; if there is none, it will look at the next line, and so forth until a permissible location is found.) You can also specify two lines of text which will appear on the menu lines when the alternate mode is entered. You can specify whether this alternate mode of operation will be a *select* or an *enter text* type of operation. If selection is chosen, the text from the cursor to the next occurrence of two adjacent spaces is highlighted. Usually for this type of operation, the only allowable cursor positions are at the beginning of each selectable item on the display, so the entire selected item is highlighted.

If you specified the *enter text* type of operation, the user may enter alphanumeric data in any permissible cursor location. You may specify the color combination which this entered data will have. When data entry is complete, the same escape character specified above is used to terminate the screen mode of operation.

Determining Permissible Cursor Positions

The heart of the menu program is the capability to specify which positions on the screen the cursor is permitted to occupy. When in the selection mode, there should only be one permissible cursor position for each item to be selected. That is usually the first character of the

item description. When changing or entering data on the screen, a file structure is usually determined, which specifies the names of the data items for each file entry and the length of each of these items. A display location and length is established for the display of each item, and the cursor is only allowed to go to the allocated space for each item. These areas of the screen can then be read to obtain the modified data. By prohibiting the cursor from going to other areas, it becomes impossible for the user to overwrite item definitions or to enter data that is too long to fit into the file.

The permissible cursor positions are controlled in the menu program by use of a bit map, which contains one bit for each character position on the screen. The bit map consists of a character array of 25 by 10 characters. The 25 is for the 25 screen lines; the 10 is for ten bytes of eight bits each to provide for the 80 character positions on a line.

If a particular bit is one, the cursor is permitted to go to that location; if it is a zero, the cursor is prohibited from going there. The bit map for a particular menu

Listing 1 — Cont'd

```
          for (indx = 0; indx < menu_second_line_length; indx++)
                  if (indx < length)
                          putcolorchar(menu_second_line[indx],
                                  highlight_color);
                  else
                          putcolorchar(menu_second_line[indx],
                                  menu_color);
  }
  else
  {
          color_printf(screen_first_line,menu_color);
          gotoxy(2,first_line_loc+1);
          color_printf(screen_second_line,menu_color);
  }
  choices = 10 * (values[0] - '0') + values[1] -'0';
  gotoxy(column,row);
  for(;;)
  {
          key_id = getch();
          if (key_id == 0)
                  key_id = getch()+256;
          if (menu_type <= 1)
          {
                  switch(key_id)
                  {
                  case 13:
                          if(choice > select_no)
                                  goto ExitPoint;
                          change_line_color(47);
                          gotoxy(2,23);
                          color_printf("%s",screen_color,spaces);
                          gotoxy(2,23);
```

Listing 1 — Cont'd

```
        color_printf("%s",menu_color,
        screen_first_line);
        gotoxy(2,24);
        color_printf("%s",screen_color,spaces);
        gotoxy(2,24);
        color_printf("%s",menu_color,
                screen_second_line);
        gotoxy(column,row);
        menu_type +=2;
        break;
case 333:    /*Right Arrow*/
        choice = choice + 2;
case 331:  /*Left Arrowb*/
        --choice;
        if (choice < 1)
                choice = choices;
        if (choice > choices)
                choice = 1;
        start = 10 * (values[(choice-1)*4+2] - '0')
                +values[(choice-1)*4+3] - '0';
        length = 10 * (values[(choice-1)*4+4] - '0')
                +values[(choice-1)*4+5] - '0';
        gotoxy(2,24);
        for (indx = 0;indx < menu_second_line_length;
                indx++)
        {
                if ((indx >= start) && (indx < start
                + length))
                        putcolorchar
                                (menu_second_line
                                [indx],
                                highlight_color);
                else
                        putcolorchar
                                (menu_second_line
                                [indx],
                                menu_color);
        }
        gotoxy(column,row);
        break;
default:
        if ((key_id >= 0x41) && (key_id <= 0x7A))
        {
                temp = toupper(key_id);
                for (indx=0; indx<=choices; indx++)
                {
                        start = 10 * (values[
                                (indx-1)*4+2] -
                                '0')+values[(indx
                                -1)*4+3] - '0';
                        if (temp == menu_second_line
                                [start])
                        {
                                choice = indx;
                                if(choice >
                                      select_no)
                                      goto ExitPoint;
                                change_line_color
                                (47);
                                gotoxy(2,23);
                                color_printf("%s",
                                screen_color,spaces);
                                gotoxy(2,23);
                                color_printf("%s",
                                menu_color,
                                screen_first_line);
                                gotoxy(2,24);
                                color_printf("%s",
                                screen_color,spaces);
                                gotoxy(2,24);
                                color_printf("%s",
                                menu_color,
                               screen_second_line);
                                gotoxy(column,row);
                                menu_type += 2;
                        }
                }
```

may be generated dynamically or it may be generated manually by determining what bits need to be set and where they are located in the array. Since this latter process can be rather cumbersome, a utility function, *set_cursor*, has been provided to do the job automatically. You temporarily insert *set_cursor* into your program, just after the display has been generated. You can now move the cursor around the display and insert an *x* or *X* at every position where the cursor is to be allowed. You should also replace any *x* or *Xs* that occur naturally in the display in locations prohibited to the cursor with some other character. Make sure not to hit the *Enter* key until you have inserted all of the required *x*s in the display.

When you hit *Enter* the program reads the entire screen and generates a file called *MATRIX.C*, which contains all of the ASCII data needed for initializing a cursor map array in your program. You can set up the bit map array by inserting the following line in your program:

```
char nnnnnnnn[25][10] =
```

where nnnnnnnn is the name of your bit map. If you are using the Turbo C total environment editor, place the cursor after the equals sign and type *^KR*. When asked for the file name, type in *MATRIX.C*. The required data for the bit map will then be inserted into your program. You can then remove *set_cursor* from your listing and recompile the program. When you run the menu function, you will find that the cursor will only go to those positions which you marked with an *x* or *X* when you used *set_cursor* to construct the bit map.

Key Designations

Normal keys on the IBM PC keyboard generate the standard ASCII representation of the selected letter or number. Most special keys, such as the cursor arrow keys and the F1 through F10 function keys return two characters, first a hex 0 and then some number from 1 to 255. To put all key returns into a common format, the menu program automatically reads a second character from the keyboard when the first character is 0 and adds 256 to this second character to obtain a unique

Listing 1 — Cont'd

```
                }
            }
    }
    else
    {
            if (key_id == escape_char)
                    goto ExitPoint;
            switch(key_id)
            {
                    case 8:    /*Backspace*/
                    case 331:  /*Left Arrow*/
                            if ((menu_type == 0) ||
                            (menu_type == 2))
                                    change_line_color
                                    (screen_color);
                            do
                            {
                                    column--;
                                    if (column < 0)
                                    {
                                            column = 79;
                                            row --;
                                            if (row < 0)
                                                    row = 24;
                                    }
                                    indx = column/8;
                                    remainder = column - indx*8;
                                    interim = map[row][indx] &
                                    (0x01 << remainder);
                            }
                            while   (interim == 0x00);
                            gotoxy(column,row);
                            if ((menu_type == 0) ||
                            (menu_type == 2))
                                    change_line_color
                                    (highlight_color);
```

Listing 1 — Cont'd

```
            break;
    case 333:  /*Right Arrow*/
            if ((menu_type == 0) ||
            (menu_type == 2))
                    change_line_color
                    (screen_color);
            do
            {
                    column++;
                    if (column > 79)
                    {
                            column = 0;
                            row ++;
                            if (row > 24)
                                    row = 0;
                    }
                    indx = column/8;
                    remainder = column - indx*8;
                    interim = map[row][indx] &
                            (0x01 << remainder);
            }
            while   (interim == 0x00);
            gotoxy(column,row);
            if ((menu_type == 0) ||
            (menu_type == 2))
                    change_line_color
                    (highlight_color);
            break;
    case 13:
            column = 0;
    case 336:  /*Down Arrow*/
            if ((menu_type == 0) ||
            (menu_type == 2))
                    change_line_color
```


number that will not duplicate one of the normal ASCII codes. This key input is stored in a variable called *key_id*. Thus, when looking at the program listing, some of the comparisons of *key_id* with various numbers may appear unfamiliar. They can be identified by taking any chart of keyboard codes and adding 256 to the second character generated by a particular key. The menu program has the flexibility of specifying which key will be used to escape from the screen type of operation. This capability will be described in furthur detail below. You should note, however, that whatever character you select for escape cannot be entered as data on the screen.

Menu Options

The programmer has almost unlimited flexibility in defining how the menu program is to be used. The menu options are selected by parameters passed through the menu function.

The first parameter passed to the menu function is a string called *values* which defines the characteristics of the menu line. It begins with two digits which define the number of menu function items. Four digits then follow for each menu item. The first two represent the column of the display at which that menu item begins. The next two represent the number of characters in the menu item. These four digit entries must correspond to the actual spacing of the entries in the menu line string described below.

The next parameter, called *menu_first_line*, is a string showing the first line of instructions when the program is showing the menu type display. Following that is a string called *menu_second_line*, which is the second line of instructions for the menu type (the actual set of menu function items). Next are two strings called *screen_first_line* and *screen_second_line*, which are the first and second lines of instructions when the menu program switches to the second type of operation.

The next parameter, called *menu_type*, selects the mode of operation. There are four modes of operation for the menu function, which are

Listing 1 — Cont'd

```
                (screen_color);
        row++;
        if (row > 24)
                row = 0;
        indx = column/8;
        remainder = column - indx * 8;
        interim = map[row][indx] & (0x01 <<
        remainder);
        if (interim != 0x00)
        {
                gotoxy(column,row);
                if ((menu_type == 0) ||
                (menu_type == 2))
                        change_line_color
                        (highlight_color);
                break;
        }
        column = 0;
        do
        {
                indx = column/8;
                remainder = column - indx*8;
                interim = map[row][indx] &
                (0x01 << remainder);
                if (interim != 0x00)
                {
                        gotoxy(column,row);
                        if ((menu_type == 0)
                        || (menu_type == 2))
                           change_line_color
                           (highlight_color);
                        break;
                }
                column++;
                if (column > 79)
                {
                        column = 0;
                        row ++;
                        if (row > 24)
                                row = 0;
                }
        }
        while   (interim == 0x00);
        break;
case 328:  /*Up Arrow*/
        if ((menu_type == 0) ||
        (menu_type == 2))
                change_line_color
                (screen_color);
        row--;
        if (row < 0)
                row = 24;
        indx = column/8;
        remainder = column - indx * 8;
        interim = map[row][indx] & (0x01
        << remainder);
        if (interim != 0x00)
        {
                gotoxy(column,row);
                if ((menu_type == 0) ||
                (menu_type == 2))
                        change_line_color
                        (highlight_color);
                break;
        }
        column = 79;
        do
        {
                indx = column/8;
                remainder = column - indx*8;
                interim = map[row][indx] &
                (0x01 << remainder);
                if (interim != 0x00)
                {
                        gotoxy(column,row);
                        if ((menu_type == 0)
```

Listing 1 — Cont'd

```
                                          || (menu_type == 2))
                                             change_line_color
                                             (highlight_color);
                                          break;
                                    }
                                    column--;
                                    if (column < 0)
                                    {
                                          column = 79;
                                          row --;
                                          if (row < 0)
                                                row = 24;
                                    }
                              }
                              while   (interim == 0x00);
                              break;
                        default:
                              if ((menu_type == 1) ||
                              (menu_type == 3))
                              {
                                    putcolorchar(key_id,
                                    screen_color);
                                    do
                                    {
                                          column++;
                                          if (column > 79)
                                          {
                                                column = 0;
                                                row ++;
                                                if (row > 24)
                                                      row =
                                                0;
                                          }
                                          indx = column/8;
                                          remainder = column -
                                          indx * 8;
                                          interim = map[row]
                                          [indx] & (0x01
                                          << remainder);
                                    }
                                    while   (interim == 0x00);
                                    gotoxy(column,row);
                        }
                  }
            }
      }
      ExitPoint:
      return choice;
}

/*
change_line_color = changes the color of a line up to a double space
*/

void change_line_color(int color)
{
      union REGS rin;
      char prev_char;

      prev_char = 's';
      for( ; ; )
      {
            rin.h.ah = 3;
            rin.h.bh = 0;
            int86(0x10,&rin,&rin);
            ch = read_char_from_screen();
            if((ch == ' ') && (prev_char == ' '))
                  break;
            prev_char = ch;
            gotoxy(rin.h.dl,rin.h.dh);
            putcolorchar(ch,color);
      }
      gotoxy(column,row);
}

/*
clearscreen = clears the screen and displays selected color background
```

controlled by this parameter. They are assigned by one of the numbers zero to three. The modes of operation are:

0 = When the menu program is called, it will start with the menu display. When it is in the screen display type of operation, all alphanumerics will be ignored.

1 = When the menu program is called, it will start with the menu display. When it is in the screen display type of operation, all alphanumerics will be displayed where typed.

2 = When the menu program is called, it will start with the screen display. When it is in the screen display type of operation, all alphanumerics will be ignored.

3 = When the menu program is called, it will start with the screen display. When it is in the screen display type of operation, all alphanumerics will be displayed where typed.

The next parameter, *screen_color*, is the screen color. This has no effect upon the display that has already been generated, but does determine what color combination will be used for alphanumerics typed on the screen and to restore the background for that part of the two instruction lines that is not used. Normally it should be the same color combination used in generating the original display screen. Table 1 shows the color combinations represented by each number. The next parameter, called *menu_color*, is the color combination used by the two lines of instructions produced by the menu. The next parameter, called *highlight_color*, is the color combination used to highlight the selected menu item.

The next parameter determines how many menu functions will switch to the screen type of operation when selected. Those functions which switch to the screen type of operation must be grouped at the beginning of the menu line, since the count in this variable begins with the leftmost function.

The next parameter, called *escape_char*, is the representation of the key which must be hit to escape from the screen type of operation. It is a number which is the ASCII value produced by a regular key or the value plus 256 if the keyboard output is a two character output beginning with 0. Thus any keyboard output may be selected.

Listing 1 — Cont'd

```
*/

void clearscreen(int color)
{
        int indx;
        union REGS reg;

        gotoxy(0,0);
        reg.h.ah = 9;
        reg.h.al = 0x20;
        reg.h.bh = 0;
        reg.h.bl = color;
        reg.x.cx = 2000;
        int86(0x10,&reg,&reg);
}

/*
color_printf = printf with selected foreground and background colors
*/

void color_printf (char *msg,int color,...)
{
        union REGS reg;
        char ch, string[2000];
        int i = 0;
        va_list (ap);

        va_start (ap,msg);
        vsprintf(string,msg,ap);  /*do printf to string*/
        va_end(ap);
        while ((ch=string[i++]) != '\0')   /*get chars from string till end*/
        {
                if (ch == 0x0A)   /*is character a line feed*/
                {
                        reg.h.ah = 3;
```

Listing 1 — Cont'd

```
                        int86(0x10,&reg,&reg);
                                /*get cursor position*/
                        reg.h.dl = 0;
                        reg.h.dh++;
                                /*cursor value to beginning of next line*/
                        reg.h.ah = 2;
                        int86(0x10,&reg,&reg);
                                /*set new cursor position*/
                }
                else
                {
                        reg.h.ah = 9;
                        reg.h.al = ch;
                        reg.x.bx = color;
                        reg.x.cx = 1;
                        int86(0x10,&reg,&reg);
                                /*send a color character to display*/
                        reg.h.ah = 3;
                        int86(0x10,&reg,&reg);
                                /*get cursor position in D reg*/
                        reg.x.dx++;
                        reg.h.ah = 2;
                                /*increment cursor position value*/
                        int86(0x10,&reg,&reg);
                                /*set cursor to new position*/
                }
        }
}

/*
gotoxy = moves cursor to selected column and row
*/

void gotoxy(int col, int row)
{
        union REGS reg;
        reg.h.ah = 2;
        reg.h.bh = 0;
        reg.x.dx = (row << 8) +col;
        int86(0X10,&reg, &reg);
}

/*
putcolorchar =  displays a character with selected color foreground
                and background
*/

void putcolorchar(char character, int color)
{
        union REGS reg;
        reg.h.ah = 3;
        reg.h.bh = 0;
        int86(0x10,&reg,&reg);
        reg.h.ah = 9;
        reg.h.al = character;
        reg.h.bl = color;
        reg.x.cx = 1;
        int86(0x10,&reg,&reg);
        reg.h.ah =2;
        reg.h.dl = reg.h.dl+1;
        int86(0x10,&reg,&reg);
}

/*
read_char_from_screen = reads a character from the screen into 'ch'
*/

char read_char_from_screen()
{
        char ch;
        union REGS reg;

        reg.h.ah = 3;
        reg.h.bh = 0;
        int86(0x10,&reg,&reg);
```

When this key is hit, it will immediately cause an exit from the screen mode of operation. The next parameter, called *first_line_loc*, determines the line on which the first of the two menu lines begins. It may be any screen line from 0 to 23. Normally the menu lines should be either at the top or bottom of the screen.

The final parameter is the address of the bit map, which determines which are the permissible positions of the cursor. The bit map has already been described above.

Supporting Functions

The menu function uses several supporting functions. These include *clearscreen*, which clears the screen by filling it with spaces of a designated color; *gotoxy*, which positions the cursor at a desired column and row; *putcolorchar*, which displays a character at the cursor location with a specified color combination and moves the cursor to the next column; *color_printf*, which acts like the standard C *printf* function except that it displays its data with a selected color combination; and *change_line_color*, which changes the color of a line of characters from the current cursor position up until a double space is encountered. Listings of these functions are shown for completeness. Many of them may already be available in standard libraries, but they are not included with the current version of Turbo C. If you are adapting the menu function for a monochrome display, standard monochrome equivalent functions may be used in place of some of these functions.

Conclusions

The menu function provides a great deal of flexibility in manipulating data and making menu selections. About the only restriction is that all menu function item names must fit on one line. Colors, instructions, titles, order of mode, cursor settings, and whether or not alphanumerics are to be displayed are all under the control of the programmer through the manner in which he calls the menu function. □

Listing 1 — Cont'd

```
        reg.h.ah = 8;
        int86(0x10,&reg,&reg);
        ch = reg.h.al;
        attr = reg.h.ah;
        reg.h.ah =2;
        reg.h.dl = reg.h.dl+1;
        int86(0x10,&reg,&reg);
        return ch;
}

/*
set_cursor = sets up array of permissible cursor positions
*/

void set_cursor()
{
        int i,j,indx,remainder,key_value;
        char interim,map[25][10];

        FILE *f1;
        f1 = fopen("matrix.c","w");

        for(i=0;i<=24;i++)
        {
                for(j=0;j<=9;j++)
                {
                        map[i][j]=0x00;
                }
        }
        row=0;
        column = 0;
        gotoxy(column,row);
        while ((key_value = getch()) != 13)
        {
                if (key_value == 0)
```

Listing 1 — Cont'd

```
                key_value = getch()+128;
        switch(key_value)
        {
                case 8:   /*Backspace*/
                case 203:  /*Left Arrow*/
                        --column;
                        break;
                case 205:  /*Right Arrow*/
                        ++column;
                        break;
                case 208:  /*Down Arrow*/
                        ++row;
                        break;
                case 200:  /*Up Arrow*/
                        --row;
                        break;
                default:
                        putch(key_value);
                        column++;
        }
        if (column > 79)
        {
                column = 0;
                row++;
        }
        if (column < 0)
        {
                column = 0;
                row--;
        }
        if (row < 0)
                row = 0;
        gotoxy(column,row);
}
row=0;
column = 0;
while (row*column < 1896)
{
        gotoxy(column,row);
        ch = read_char_from_screen();
        if ((ch == 'x') || (ch == 'X'))
        {
                indx = column/8;
                remainder = column - indx * 8;
                interim = 0x01;
                interim = interim << remainder;
                map[row][indx] = map[row][indx] | interim;
        }
        column++;
        if (column > 79)
        {
                column = 0;
                row++;
        }
        gotoxy(column,row);
}

clearscreen(30);
fputc('{',f1);
gotoxy(0,0);
for (i=0;i<=24;i++)
{
        for (j=0;j<=9;j++)
        {
                fprintf(f1,"0x%x",map[i][j]);
                if ((i != 24) || (j != 9))
                        fputc(',',f1);
        }
        fprintf(f1,"\n");
}
fputc('}',f1);
fclose(f1);
}
/* End of File */
```

NEW! VERSION 3.1

COMPUTER LANGUAGE PRODUCTIVITY AWARD 1990-91

INSTALL.BAT

FIRST IMPRESSIONS LAST.

INSTALL 3.1 GIVES YOUR SOFTWARE PRODUCT A PROFESSIONAL INTRODUCTION.

Installation procedures that rely on batch files and DOS commands not only look amateurish but also have limited error handling capabilities. Today's users expect quality and ease of use from their software beginning the moment they open the package. A smooth, professional installation makes an important first impression.

NEW FEATURES

Over 50 new/enhanced keywords
Now licensed once, like a compiler
Supports products of unlimited size
INSTALL.EXE is still less than 80K
Displays no copyright banner
Supports Borland C++ 3.0

Microsoft Windows Version coming soon!

PROVEN RELIABILITY

INSTALL 3.1 has been used for over four years by some of the biggest (and smallest!) names in the industry, in the U.S. and abroad, to install millions of copies of their programs. Add your name to that list today.

INSTALL PRO

- Builds Distribution Disk Sets Automatically
- Full Screen Interface
- Creates Script Files and DISK.ID Files
- High Performance Data Compression
- Slashes Work Time Up to 75%
- Splits & Compresses Files Of Any Size
- Can Tag A File, Directory, Or Directory Tree With One Keystroke
- Can Automatically Format 360K, 720K, 1.2M, and 1.44M Disks
- Fully Compatible With Install 2.0 & Later

KNOWLEDGE DYNAMICS CORPORATION
P.O. Box 1558
Canyon Lake, Texas 78130-1558

CALL OR FAX BY 10 am (CST) AND RECEIVE INSTALL 3.1 TOMORROW

✧ Request 453 on Reader Service Card ✧

EVERYTHING YOU NEED

INSTALL 3.1 is customized for your product with an ASCII text file called a "script file". INSTALL 3.1 comes with many sample script files that you can modify and use immediately. No programming is necessary to create most installations. However, C source code is included for your convenience and technical support is free.

FAST AND SIMPLE

Use INSTALL 3.1 to create an elegant installation procedure that will increase users' confidence in your product. INSTALL 3.1 takes advantage of all available RAM for lightning-fast file transfers. All your documentation has to tell users is TYPE "A:INSTALL".

30 DAY MONEY-BACK GUARANTEE

Free technical support. No royalties. MasterCard/VISA/ COD/POs welcome.

$399.95	INSTALL 3.1 PRO
$249.95	INSTALL 3.1 Standard
$ 99.95	International Option
$ 99.95	OS/2 Option

SALES 1/800-331-2783X195

International 1/512-964-3994
24 hour FAX 1/512-964-3958
24 hour BBS 1/512-964-3929

Yet Another C++

Adolfo Di Marc

I use C to program information systems that commonly handle money quantities. Using *doubles* to represent money quantities can fail because the decimal digits are not represented correctly and round-off errors sometimes occur. A friend of mine, who uses Canon BASIC, has always used *floor(double*100.0)* to represent money quantities. Any amount is represented as a *double* with no fractional part. This trick loses no decimals because it

uses the *double* type, with its 15+ digit precision, as a compiler supported *long long*.

For example, $25.35 is represented as the *double* 2535.0, using a scale factor of 100 (for two decimal places). The problem I kept facing in my C programs was remembering when to multiply by 100 and when not to. For example, adding two *doubles* that represent money quantities doesn't require you to multiply by 100

```
$25.35 + $35.75
-> 2535.0 + 3575.0 == 6110.0
-> $61.10
```

In addition, one should never multiply by the scale factor when multiplying a money quantity

```
$100 * (1.0+0.06)
-> 10000.0 * (1.06) == 10600.0
-> $106
```

It is very easy to forget to multiply by the scale factor when using *doubles* as money. When I started programming in C++, I realized that a C++ money class was the solution for these problems.

Defining Requirements

After examining Zortech's C++ *money* class (see the sidebar on available tools), I sat down to define the requirements for my own *money* class. I came up with seven design goals:

- Money quantities should behave as regular numbers.
- The *money* class should be portable and not compiler vendor dependent.
- The programmer should be protected from misusing money quantities.
- It should be possible to use standard library functions with money quantities.
- Most operators should be inline, to let the compiler optimize the generated code.
- The money header file should be short.
- The programmer should be able to define the number of decimals in a money data item.

Money Class

I ended up having to devote more storage space to the *money* variable than I originally wanted: eight bytes for most computers, compared to six (25 percent less) for the Zortech implementation.

Listing 1 is the header file *money.h*, which defines and implements the *money* class. All the methods in this class are inline, fulfilling my fifth requirement. The various arithmetic operators cater to the diverse situations found in real-life programs.

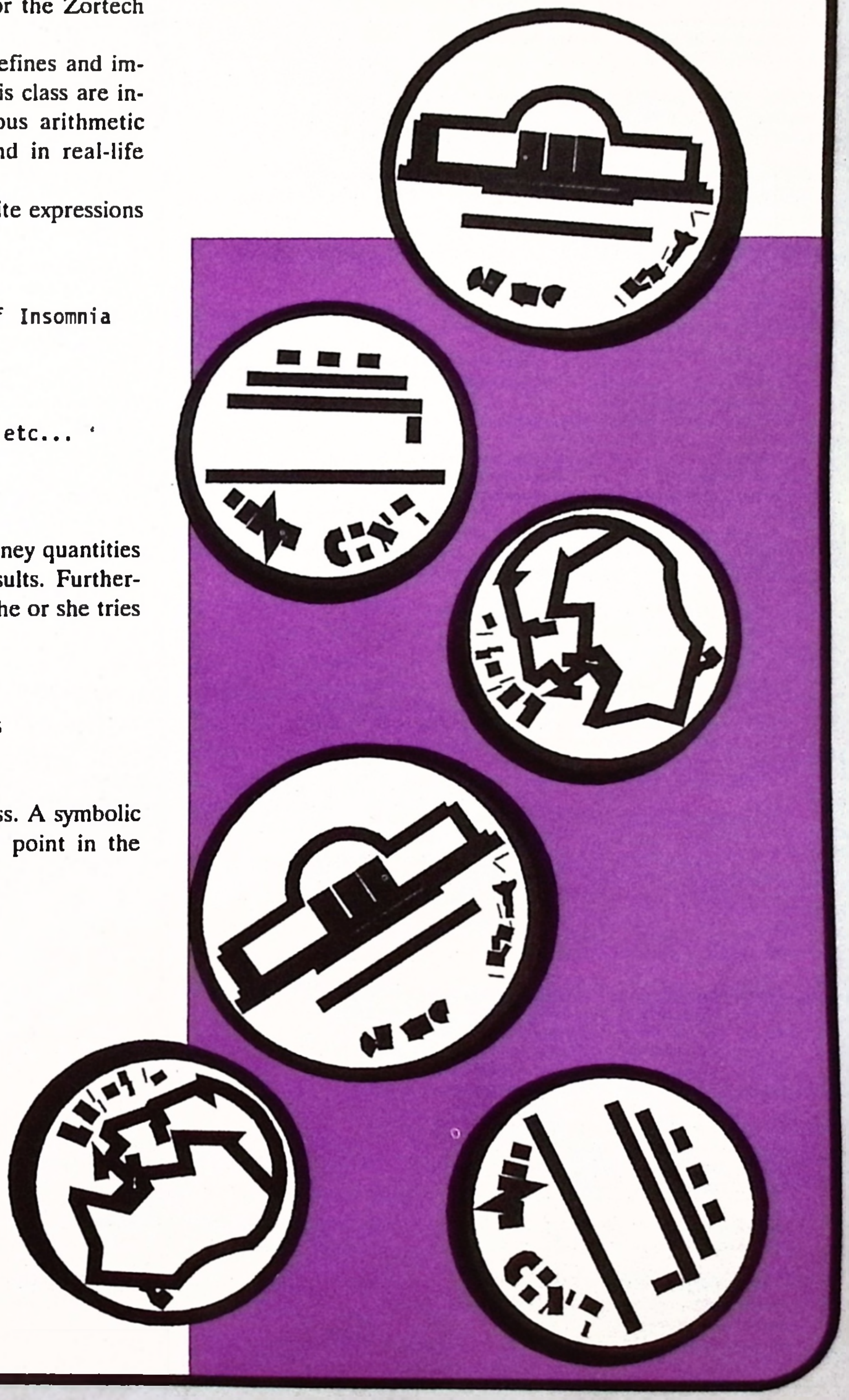
The *money* class permits a programmer to write expressions such as

```
money mm,m = 1000; // I've got a thou'
double tax = 0.23; // State Government of Insomnia
m *= (1-tax);     // This is what I have
m = m+500;     // Thanks, Mom...
mm = 500;
m = m + 1500 - (mm / m) * (1.0/3.0); // etc...
printf("Salary = %10.2f\n",
    (double) ((10+mm)/m * m));
```

In a nutshell, the programmer can freely mix money quantities with regular numbers to obtain the correct results. Furthermore, the compiler will warn the programmer if he or she tries to misuse money data items, as in

```
money m, mm;  // ok
double d = m*mm; // can't multiply moneys
mm = d/m;    // can't divide by money
```

Listing 2 is a test program for the *money* class. A symbolic debugger will illuminate the goings-on at each point in the program.



Adolfo DiMare preaches and does research in programming at the Universidad de Costa Rica. When he is not busy reading The C Users Journal, *he can be reached at (506) 24-0504 or as* adimare@OCRVM2 *through BITNET.*

Listing 1 (money.h)

```
/* @(#) money.h          Copymiddle 1991 Adolfo Di Mare  */
/*                                                       */
/*           Yet Another Money C++ Class                 */
/*                                                       */
/*  Use freely but acknowledge author and publication.  */
/*   DO NOT SELL IT. The author reserves all rigths!     */

/*                               BITNET: adimare@UCRVM2  */

/* Compiler:                          Borland C++ v 2.0 */
/*                             [should work with Turbo C++] */


#ifndef _money_h
#define _money_h

extern "C" {      // avoid type clashes with the C library
    #include <math.h>     /* floor() */
    #include <float.h>    /* DBL_DIG */
}

#ifndef MONEY_DECIMALS       /* number of decimals in */
#define MONEY_DECIMALS  2    /*   any money quantity  */
#endif                       /* don't use parentesis! */

#define __VAL(n)  n               /* 1 level indirection */
#define TENPOW(n) __VAL(1.0e##n)  /* Trick to yield 10^n */


#define MONEY_DIG   DBL_DIG


class money {
public:

    static int    decimals() { return MONEY_DECIMALS;     }
    static int    digits()   { return MONEY_DIG; }
    static double SCALE()
        { return TENPOW(MONEY_DECIMALS); }

    money();                // do nothing constructor
    money(double);          // constructor from double
    money(const money&);    // copy constructor

    money& operator= (const money&);  // copy operator
    money& operator= (double);        // copy from double
    operator double() const;          // convert to double

    int  OK()   const;  // check money's invariant
    void FIX();         // get rid of unwanted decimals


    friend money  operator + (const money&, const money&);
    friend money  operator + (double,       const money&);
    friend money  operator + (const money&, double);
    friend money  operator - (const money&, const money&);
    friend money  operator - (double,       const money&);
    friend money  operator - (const money&, double);

    friend money  operator*  (const money&, double);
    friend money  operator*  (double,       const money&);
    friend double operator/  (const money&, const money&);
    friend money  operator/  (const money&, double);
    friend money  operator%  (const money&, const money&);

    // money  * money  is NOT valid
    // double / money  is INVALID

    friend int operator == (const money&, const money&);
    friend int operator != (const money&, const money&);
    friend int operator <  (const money&, const money&);
    friend int operator >  (const money&, const money&);
    friend int operator <= (const money&, const money&);
    friend int operator >= (const money&, const money&);

    money& operator += (const money&);
    money& operator += (double);
    money& operator -= (const money&);
    money& operator -= (double);

    money& operator *= (double);
    money& operator /= (double);

    friend money  operator+ (const money&);
    friend money  operator- (const money&);
    money& operator++();
    money& operator--();

    friend int    operator! (const money&);

    friend money abs(const money&);
    friend money flatten(
        const money& m,
        double cents=0.25, int rounding = 1 /* TRUE */);

protected:         // let users change the class behaviour
    double m_money;
};

// Constructors && assignment
inline money::money() {
// do nothing constructor, for efficiency
}
inline money::money(double d) {
// construct from double
    m_money = d*SCALE();
    FIX();
}
inline money::money(const money& m) {
// copy constructor
    m_money = m.m_money;
}

inline money& money::operator= (const money& m) {
// copy operator
    m_money = m.m_money;
    return *this;
}
inline money& money::operator= (double d) {
// assign from double
    m_money = d*SCALE();
    FIX();
    return *this;
}
inline money::operator double() const {
// convert to double
    return m_money / SCALE();
}

inline int money::OK() const {
// Returns TRUE (1) when the quantity stored
// in *this really corresponds to a money
// quantity.

    money temp;
    temp.m_money = m_money;
    temp.FIX();
    return (
        ( temp.m_money == m_money )
          &&
        ( fabs(m_money) < (TENPOW(DBL_DIG) / SCALE()) )
    );
}

inline void money::FIX() {
// Deletes all decimals digits beyond
// the MONEY_DECIMALS decimal place.
// - If the value is out of range, FIX
//   won't fix it.
    m_money =
        (m_money > 0.0
            ?
              floor(
                  m_money
                  #ifdef MONEY_ROUNDING
                    + 0.5 // 0.49 is also an option...
                  #endif
              )
            :
              ceil(
                  m_money
                  #ifdef MONEY_ROUNDING
                    - 0.5
                  #endif
              )
        );
}

// add
inline money operator+ (const money& m, const money& mm) {
    money temp;    // don't mult*SCALE()
    temp.m_money = m.m_money + mm.m_money;
    return temp;
}
```

Implementation Details

I first tried to fake money quantities using *longs*, but the difficulty in doing so stopped me from pursuing this approach. (The Zortech tool's *money* type is implemented using integer arithmetic.)

After I realized that the *money* variable would be a *double*, the main implementation problem was to keep track of when to multiply by the scale factor and when not to. This I accomplished in each of the overloaded arithmetic operators.

I also implemented the *money::FIX()* member functions to get rid of the excess decimals whenever appropriate. When the preprocessor constant *MONEY_ROUNDING* is defined, the excess decimals in a *double* assigned to a *money* variable are rounded. Otherwise the excess decimals are truncated

```
money m(1.5199);   // $1.52, when
                   // MONEY_ROUNDING

money m(1.5199);   // $1.51, when
                 // not MONEY_ROUNDING
```

The programmer cannot selectively choose whether to round or not case by case because the decision is made at compile time.

The preprocessor constant *MONEY_DECIMALS* defines how many decimals a money item has. The member function *money::SCALE()* is implemented using a preprocessor trick that returns the scale factor used to multiply a *double* to make it a money quantity. If three decimals are needed for money items, the scale factor would be 1,000 = 10^3. In some countries the inflation is so high that the number of decimals is negative. In this case the scale factor would be a number less than one. Since *money::SCALE()* is an inline function, the compiler can optimize out the division by the scale factor in some cases. If the programmer doesn't define *MONEY_DECIMALS*, then the class uses a default value of two decimals. In my programs, I define *MONEY_DECIMALS* before including the *money.h* file

```
#define MONEY_DECIMALS 4 // must
               // use a decimal number
#include "money.h"    // or TENPOW
                      // bombs.
```

The vector constructor *money::money()* does not initialize a money item, because in many cases doing so would be wasteful. The programmer can consider money items as regular numbers. The compiler should be able to optimize out this constructor if it is used.

Though the class *money* implements most arithmethic operators, it does not implement the following

```
money operator* (const money&,
                 const money&),
money operator/ (const double,
                 const money&);
```

It just does not make sense to use these operators in a program. If you use them, you will get a compile time error (a dissatisfied programmer could add them to the class easily).

As arithmetic operators in *money.h* demonstrate, the class is programmed to minimize the number of times that each *double* must be scaled up by the scale factor.

The comparison operators are defined only for money items. Thus when a money data type is compared to a *double*, the compiler promotes the *double* to a *money* variable using the constructor *money::money(double)*. To prevent the promotion, the programmer should use an explicit typecast

```
double d = 15.253;   // 15.253
money m = 15.25;    // $ 15.25

if (d == m) {  // TRUE: d
               // becomes money(d)
}
if (d == (double) m) { // FALSE:
               // 15.253 != 15.25
}
```

The function *flatten(money, cents, rounding)* rounds up a money quantity to the nearest value that can be paid in coins. For example, Costa Rica has no one cent coin because the smallest coin,

Listing 1 — Cont'd

```
inline money operator+ (double d, const money& m) {
    return (money(d)+m);
}
inline money operator+ (const money& m, double d) {
    return (m+money(d));
}

// substract
inline money operator- (const money& m, const money& mm) {
    money temp;
    temp.m_money = m.m_money - mm.m_money;
    return temp;
}
inline money operator- (double d, const money& m) {
    return (money(d)-m);
}
inline money operator- (const money& m, double d) {
    return (m-money(d));
}

// multiply
inline money operator* (const money& m, double d) {
    money temp;
    temp.m_money = m.m_money * d; // don't mult by SCALE()
    temp.FIX();    // this could be delayed...
    return temp;
}
inline money operator* (double d, const money& m) {
    return (m*d);
}

// divide
inline double operator/ (const money& m, const money& mm) {
    return m.m_money / mm.m_money;
}
inline money  operator/ (const money& m, double d) {
    money temp;
    temp.m_money = m.m_money / d;
    temp.FIX();    // this could be delayed...
    return temp;
}
inline money operator%  (const money& m, const money& mm) {
    money temp;
    temp.m_money = fmod(m.m_money, mm.m_money);
    temp.FIX();    // this could be delayed...
    return temp;
}

// compare
inline int operator  == (const money& m, const money& mm) {
    return m.m_money ==  mm.m_money;
}
inline int operator  != (const money& m, const money& mm) {
    return m.m_money !=  mm.m_money;
}
inline int operator  <  (const money& m, const money& mm) {
    return m.m_money <   mm.m_money;
}
inline int operator  >  (const money& m, const money& mm) {
    return m.m_money >   mm.m_money;
}
inline int operator  <= (const money& m, const money& mm) {
    return m.m_money <=  mm.m_money;
}
inline int operator  >= (const money& m, const money& mm) {
    return m.m_money >=  mm.m_money;
}

inline money& money::operator += (const money& m) {
    m_money += m.m_money;
    return *this;
}
inline money& money::operator += (double d) {
    m_money += d*SCALE();
    FIX();
    return *this;
}
inline money& money::operator -= (const money& m) {
    m_money -= m.m_money;
    return *this;
}
inline money& money::operator -= (double d) {
    m_money -= d*SCALE();
    FIX();
    return *this;
}
inline money& money::operator *= (double d) {
    m_money *= d;
    FIX();
    return *this;
}
inline money& money::operator /= (double d) {
    m_money /= d;
    FIX();
    return *this;
}

// unary op's
inline money operator+(const money& m) {
    return m;
}
inline money operator-(const money& m) {
    money temp;
    temp.m_money = -m.m_money;
    return temp;
}
inline money& money::operator++() {
    m_money += SCALE();
    #if (MONEY_DECIMALS<0)
        FIX(); // avoid problems because of
    #endif    // the representation of 10^-n
    return *this;
}
inline money& money::operator--() {
    m_money -= SCALE();
    #if (MONEY_DECIMALS<0)
        FIX();
    #endif
    return *this;
}
inline int operator!(const money& m) {
    return m.m_money == 0.0;
}

inline money abs(const money& m) {
    money temp;
    temp.m_money = fabs(m.m_money);
    return temp;
}

money flatten(const money& m, double cents, int rounding) {
// Returns a money data item where the cents are
// rounded modulo "cents". In this way cents can
// be stripped of money items when the currency
// does not have all the coins required to pay
// every posible quantity.
    money temp;
    double c = floor(fabs(cents*money::SCALE())); // cents
    double r = fmod(m.m_money, c);              // remainder
    temp.m_money =
        (!rounding || (2.0* r <= c)
            ? m.m_money - r
            : m.m_money - r + c
        );
    return temp;
}

/* Avoid name space overcrowding */
#undef __VAL
#undef TENPOW /* jic: Just In Case! */

#endif  /* _money_h */

/* End of File */
```

called a *peseta*, is worth 25 cents of a *colón*. In the following example the money item *m* is rounded up to *pesetas*

```
money m = 125.80;   // 125.88 colones
money mm = flatten(m); // 125.75 colones
```

The header file called *money.h* implements the complete *money* class using inline functions, which permit the compiler to optimize out any floating point operations it can. I decided not to provide more operators for money items, because the type converter *money::operator double()* allows the standard math functions to work with money items. *money.h* includes only two files, *math.h* and *float.h*, which contain the prototypes for the functions *floor()*, *ceil()*, *fmod()*, and the constant *DBL_DIG*. In order to use bigger money quantities, you can declare every *double* variable in *money.h* as a *long double*.

As implemented, the money class should be quite portable because it does not make use of any odd C++ constructs. Depending on the compiler used, it is quite possible to optimize out many of the inline operators, and thus yield efficient programs.

Conclusion

This tiny C++ money class lets the programmer use money items with ease. The implementation is as efficient as using floating point values in arithmetic expressions. This should be reason enough to use it right away. □

Listing 2 (money.c)

```
/* @(#) money.c                          1991 Adolfo Di Mare  */
/*                                        adimare@UCRVM2     */
/*                                                           */
/*                  Test driver for money.h                  */
/*                                                           */
/* Compiler:                             Borland C++ v 2.0 */
/*                             [should work with Turbo C++] */

/*
   To see what is going on, you need to use your symbolic
   debugger to examine each of the declared variables.

   For Borland C++, I used the following watches:

        m_money      d,f18
        m,r          i
        mm,r         i
        elapsed      s,f18

   Change the compile time macros to see how money's
   change their behaviour.

*/

#define TEST

#ifdef  TEST /* ( */

#if 0
   #define MONEY_ROUNDING      /* Force rounding of doubles */
#endif

#define MONEY_DECIMALS  2      /* 2 decimals for money data */

extern "C++" {
   #include "money.h"
```

Listing 2 — Cont'd

```
    #ifdef __TURBOC__
    #include <iostream.h>
    #else
    #include <stream.h>
    #endif
}

inline ostream& operator<<(ostream &out, money& m) {
    return out << (double) m;
}

extern "C" {
    #include <stdlib.h>
    #include <time.h>
    #include <stdio.h>
}

#define RANGER
#undef  RANGER            /* takes forever to run...   */

void ranger(void);

int main(void) {

    // To the right is the assigned value

    money m(25.8499); // $  25.84
    money mm;         // $ #$$#@$@.98

    int     i = 5;    //     5
    long    l = 6;    //     7
    double  d = m;    //     25.8399999999999999999999999999

    i  = m;           //     25
    m  = d;           // $  25.84
    m  = 7.009999999; // $   7.oo
    m += 1;           // $   8.oo
    m += 1.245000001; // $   9.24
    m += 1L;          // $  10.24
    m += 'a';         // $  'a'+10.24

    m  = -d;          // $ -25.84
    m  = -7.00999999; // $ - 7.oo
    m -= 1;           // $ - 8.oo
    m -= 1.245000001; // $ - 9.24
    m -= 1L;          // $ -10.24
    m -= 'a';         // $ -'a'-10.24

    mm = 10;          // $  10.oo
    m  = mm+4;        // $  14.oo
    m  = mm+4.014999; // $  14.01
    m  = 4.99+mm;     // $  14.99
    m  = 4+mm;        // $  14.oo
    m  = m+mm;        // $  24.oo
    m += m;           // $  48.oo

    d  = mm+4;        //    14.oo
    d  = mm+4.011;    //    14.0099999999999998
    d  = 4.99+mm;     //    14.9900000000000002
    d  = 4+mm;        //    14.oo

    mm = 10;          // $  10.oo
    m  = mm*4;        // $  40.oo
    m  = mm*4.0;      // $  40.oo
    m  = 4*mm;        // $  40.oo
    m  = 4.0*mm;      // $  40.oo

    mm = 10;          // $  10.oo
    d  =  7.00001;    //     7.00000999999999962
    m  = d*mm+d;      // $  77.oo
    m  = (d*mm)+d;    // $  77.oo
    m  = d+d*mm;      // $  77.oo
    m  = d+(d*mm);    // $  77.oo

    m = 10;           // $  10.oo
          mm = 77;    // $  77.oo
    m = m % mm;       // $  10.oo
    // $10 == 0L * $77 + [$10]

    m = 77;           // $  77.oo
          mm = 10;    // $  10.oo
    m = m % mm;       // $   7.oo
    // $77 == 7L * $10 + [$7]

    m++;              // $   8.oo
    m-;               // $   7.oo

    m   = 11.75;                // $  11.75
    m  +=  0.12;                // $  11.87
    mm  = flatten(m,0.25,1);    // $  11.75

    m  += 0.01;                 // $  11.88
    mm  = flatten(m,0.25,1);    // $  12.oo

    m   = 11.75;                // $  11.75
    m  +=  0.12;                // $  11.87
    mm  = flatten(m,0.25,0);    // $  11.75

    m  += 0.01;                 // $  11.88
    mm  = flatten(m,0.25,0);    // $  11.75

    m  -= 5;
    m  += 0.12;                 // $   7.oo
    if (m == 0 || 0 == m) {     // nep
        m += d;
    }
    else if (!(m == m)) {       // nep
        m = m;
    }
    else if (m > m) {           // nep
        m = m;
    }
    else if (m < m) {           // nep
        m = m;
    }
    else if (m != m) {          // nep
        m = m;
    }
    else if (m >= m) {          // yep
        m *= 11;      // $  77.oo
        m += 15;      // $  92.oo
    }
    l  = i = m;       //     92
    m  = -m;          // $ -92.oo
    mm = i*l;         // $   8,464.oo
    m  = m % mm;      // $ -92.oo

    d  = 15.253;      //    15.253
    m  = 15.25;       // $ 15.25

    if (d == m) {           // TRUE:  d becomes money(d)
        l = 0;              // l = 0L
    }
    if (d == (double) m) { // FALSE: 15.253 != 15.25
        l++;                // l == 0L
    }

    // simulate a TAX calculation
    m = 0.0;
    for (i = 1; i <= 100; i++) {
        d  = i*1.005;     // 0.05% tax
        m += d;
    }                     // $ 5,075.oo
    mm = 100;             // $   100.oo
    m /= (double) mm;     // $    50.75
    m /=   3;             // $    16.91
    m *=   3;             // $    50.73

    mm = mm/mm;           //        1.00
    d  = m * (m / mm);    //     2573.5300000000002

    d  = 1.0/3.0 * m;     //       16.9100000000000001
    mm = 1.0/3.0 * m;     // $     16.91
```

Available Tools

My first move in creating a *money* class was to examine the available tools to handle money quantities. I looked at the following:

Borland's C++ *BCD* class
Zortech's C++ *BCD* class
Zortech's C++ *money* class

BCD stands for *Binary Coded Decimal*. The binary representation used in computers is a sequence of binary "digits" called *bits*. *BCD*, on the other hand, stores a number's value in base 10 digits, usually using their binary values. For instance, the number 1234 is stored in a pair of bytes using the same bit pattern as the *0x1234* hexadecimal constant. Since most computers use eight-bit bytes, each byte will hold two *BCD* digits. Many bit patterns are not valid when interpreted as *BCD* numbers: *0xFFFF*, *0x0A0A*, *0x123A*, are all invalid *BCD* quantities because in decimal notation we can use only the digits 0...9.

I chose not to use Borland's *BCD* class because it does not provide specific support to handle money quantities. For example, whenever you multiply and divide two money variables, you must figure out what to do with the digits that go beyond the number of cents. The header file in Borland C++ v2.0 pulls in the `iostream.h` header file, which slows down compilation quite a bit. I still don't use streams, and I don't like to be forced to use them if I don't have to. Also, much of the *BCD* library must be loaded when using *BCD*s, increasing the size of executable files even more. Finally, Borland doesn't include the source code for this class with the compiler, though it is available at extra charge. I have learned to examine a library's source code before using it. Zortech's *BCD* class shares most of these inconveniences.

The Zortech C++ Tools package implements a *money* class, where a money quantity is a two member structure:

```
class money
{   long dollars;
    int cents;  // ... };
```

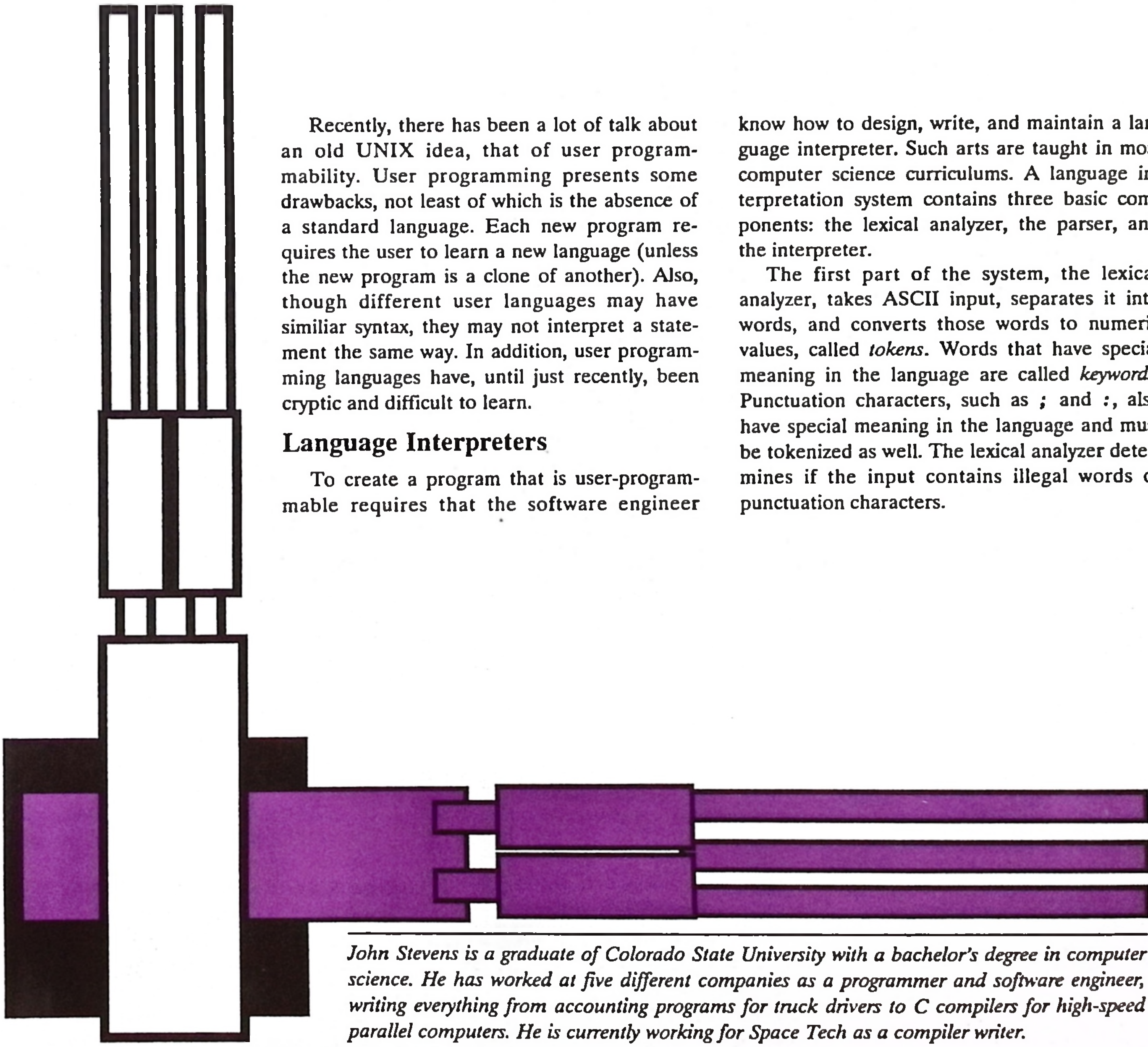
In most implementations, a *long* can hold only nine or ten decimal digits. The range of numbers is limited compared to that granted by *doubles*. This class is efficient because the operators are implemented using integer arithmetic and each money variable is quite short. □

Listing 2 — Cont'd

```
mm = 3 * mm / (3 * mm);                      // $1.oo
mm = M_PI * mm / (M_PI * mm);                // $1.oo
mm = M_PI;                                   // $3.14

mm = mm/mm + 1 - (3 * mm / (3 * mm));        // $1.oo
m *= M_E * mm / (mm * M_E) - 1;              // $0.oo

// m == 0.0 && mm == $1.oo
for (i = 1; i <= 100; i++) {
    mm /= 3;              // $    0.33
    m   = m+mm;           // Add a third
    mm  = 1;
}                         // $    33.oo

d  = m;                   //      33.00
mm = m / 330;             // $     0.10

clock_t now;
double  elapsed;

// time statistics, on an 33MHz 386
m = 0;
now = clock();
for (l = 0; l <= 100001; l++) {
    m += 1.01;              // Add $1.01
}
elapsed = (clock()-now) / CLK_TCK;
d  = elapsed;               // 3.51 secs

m = 1;
now = clock();
for (l = 0; l <= 100001; l++) {
    m *= 1.0001;            // Mult
}
elapsed = (clock()-now) / CLK_TCK;
d  = elapsed;               // 3.24 secs

m = 1;
mm = pow(10, 6);
now = clock();
for (l = 0; l <= 100001; l++) {
    mm = m;
    m /= 0.99001;           // Div
    if (!m.OK()) {
        m.FIX();            // won't fix overflows
    }
}
elapsed = (clock()-now) / CLK_TCK;
d  = elapsed;               //  8.46 secs

d  = m % m + 33;            //   33.00
m  = d;                     // $ 33.oo
mm = d / 330;               // $  0.10

d = (10+mm)/m * m;          //   10.089...

// Must use (double) type cast
printf ("Salary = %10.2f\n",
     (double) ((10+mm)/m * m));
cout <<"Salary = "
     << (double) ((10+mm)/m * m) << '\n';


// valid only if you define
// ostream& operator<< (ostream&, money&)
cout <<"Salary = "
     << (10+mm)/m * m << '\n';

    m = d / m;          // should not compile...
//  m = m * m;          // won't compile: AMBIGUITY ERROR!!!

#ifdef  RANGER
    ranger();
#endif // RANGER

    exit(0);
}


void ranger(void) {
/*
    Shows that indeed a double can hold up to
    DBL_DIG digits of precission.
*/
    // This should take forever to calculate...
    char view[] = "0123456789/123456";
    double s,t;
    double tenpow,inc;
    int    i;

    tenpow = pow(10, DBL_DIG); // 10^15
    inc    = 100.0;            // pick yours

    s = floor(tenpow+inc); // 1,000,000,000,000,000 + inc
    for (;;) {
        t = s;
        s += inc;
//      i = (int) log10(t-tenpow);
        if (s == t) {
            i = (int) log10(t-tenpow);
            cout << "BOOM t   = " << t       << '\n';
            cout << "BOOM s   = " << s       << '\n';
            cout << "BOOM inc = " << s       << '\n';
            cout << "BOOM i   = " << i       << '\n';
            view[i] = 0;
            cout << "view     = " << view    << '\n';
            return;
        }
    }

    return;
} // ranger

#endif  /* TEST */  /* } */

/* End of File */
```

Lexical Analysis Using Search Tries

John W. M. Stevens



Recently, there has been a lot of talk about an old UNIX idea, that of user programmability. User programming presents some drawbacks, not least of which is the absence of a standard language. Each new program requires the user to learn a new language (unless the new program is a clone of another). Also, though different user languages may have similiar syntax, they may not interpret a statement the same way. In addition, user programming languages have, until just recently, been cryptic and difficult to learn.

Language Interpreters

To create a program that is user-programmable requires that the software engineer know how to design, write, and maintain a language interpreter. Such arts are taught in most computer science curriculums. A language interpretation system contains three basic components: the lexical analyzer, the parser, and the interpreter.

The first part of the system, the lexical analyzer, takes ASCII input, separates it into words, and converts those words to numeric values, called *tokens*. Words that have special meaning in the language are called *keywords*. Punctuation characters, such as **;** and **:**, also have special meaning in the language and must be tokenized as well. The lexical analyzer determines if the input contains illegal words or punctuation characters.

John Stevens is a graduate of Colorado State University with a bachelor's degree in computer science. He has worked at five different companies as a programmer and software engineer, writing everything from accounting programs for truck drivers to C compilers for high-speed parallel computers. He is currently working for Space Tech as a compiler writer.

Figure 1

```
                          a , b , c , u
                          |   |   |   |
  ------------------------+   |   |   +------------------------
  |                ------------+   +------------               |
  |                |                           |               |
b , p              e                           a               s
|   |              |                           |               |
l   e , p        a , e                         p               e
|       |        |                             |               |
e       e        r                           a , e             r
        |        |                           |   |
        n        i                           b   r
        |        |                           |
        d        n                           l
                 |                           |
                 g                           e
```

Example Search Trie

While the lexical analyzer breaks the input text into words, it does not determine whether the words are arranged into legal sentences. This is the job of the parser. The parser takes a stream of tokens from the lexical analyzer and attempts to determine if they form a stream of legal sentences according to the language's grammar. A grammar consists of a set of rules that describe all legal sentences possible in the language. Not all legal sentences make sense. In most programming languages, the parser will accept legal sentences that the interpreter cannot understand or execute.

Once the parser has decided that the token stream forms legal sentences, the interpreter combines the operations of semantic analysis and program execution. Semantic analysis determines what operations the program is telling the interpreter to execute. Execution is the operation of reading progam tokens and translating them into a series of machine language function calls that instruct the CPU what to do. Interpreted languages execute slower than compiled languages, in part, because the interpreter must translate each program sentence into machine language every time it is executed. Sentences from compiled programs are already translated into machine language.

Lexical Analyzers And Search Tries

To facilitate the construction of lexical analyzers, I use a special class of search tree, called a *trie*. A trie is a tree data structure that allows strings with similiar character prefixes to use the same prefix data and store only the tails as separate data. One character of the string is stored at each level of the tree, with the first character of the string stored at the root, and the last character of the string stored at a sub-tree node or in a leaf node. Figure 1 shows how a trie would store the following collection of words: ape, append, able, bee, bearing, cape, caper, capable, us, use and user.

Tries used for lexical analysis store token values with each character in the trie, as shown in Listing 1. Most of the token values are zero, indicating `ILLEGAL KEYWORD`. The token value of the last character of each legal word is the token value for that word. For example,

in Figure 1, the *struct* for the letter *e* in the trie branch under the letter *u* stores the token value for the word *USE*. The letters *s* and *r* in the same branch would have the tokens *US* and *USER* stored with them, respectively.

Using a trie in a lexical analyzer combines the operations of breaking the input text into words and determining whether or not the words are legal for the language. This scheme imposes language design constraints on the engineer, since words do not have to be delimited to be recognized. The constraint is either to design a requirement for delimitation into the language definition or to ensure that no two adjacent words of the language can ever be combined to make a longer, legal word of the language.

Static Tries In C Arrays

I prefer to store my tries as static data in the same file as the code for the lexical analyzer. This arrangement allows each program to use more than one lexical analyzer, as well as eliminates the need for external files. On the other hand, tries have a widely variable number of elements per trie level, making it imperative to use a dynamically-sized data structure. The method I adopted stores each level of the trie as a uniquely named array with

Figure 2

```
1)  Read a character from the file.
    Call step 2 with the pointer to the root of the trie, the
    character read and a pointer to the begining of the key word
    save buffer.

2)  Search for the input character in the current node of the trie.
    If the input character is found in the trie then
        Save the input character in the key word save buffer.
        Attempt to read a character from the file.
        If End of File then
            Return End of File.
        If the character found has a child trie pointer then
            Call Step 2 with the child trie pointer, the character
            read from the file and a pointer to the next byte in the
            key word save buffer.
        If the return value from the call to step 2 is NOT FOUND then
            Unread the character read from the file
            Return the token value stored with the input character
            of this call.
        else
            Return the return value of the recursive call.
    else If the character is not found then
        Save a NUL in the key word save buffer.
        Return the value NOT_FOUND.
```

Listing 1

```
typedef struct  key_st  {
    char    c;                  /* String character.          */
    TKNS    token;              /* Token value.               */
    struct  key_st  *child;     /* Pointer to sub-trie.       */
} NODE;
```

NODE Structure

Listing 2

(	L_PAREN
)	R_PAREN
,	COMMA
/	F_SLASH
action	ACTION
after	AFTER
and	AND
archive	ARCHIVE
attributes	ATTRIBUTES
before	BEFORE
directory	DIRECTORY_T
exec	EXEC
files	FILES
hidden	HIDDEN
label	LABEL
modified	MODIFIED
name	NAME
not	NOT
or	OR
print	PRINT
readonly	READONLY
recurs	RECURS
search	SEARCH
select	SELECT
system	SYSTEM
{	L_BRACE
|	BAR
}	R_BRACE

elements corresponding to the structure in Listing 1.

Storing tries as source code in the lexical analyzer can make for very large source files. Even a small language can have a search trie that is 1100 lines of source code. Such a file compiles to a relatively small amount of data, but just as no program is ever fast enough, no program is ever small enough either. Roughly half to two-thirds of the memory that a trie uses is for storing pointers to sub-tries. To minimize the memory requirements of a search trie after compilation, you should exploit your compiler's options to group data together. Doing so lets you use a smaller pointer size to reference that data.

In order to make creating and maintaining lexical analyzers that use tries easier, I've written a program that accepts a text file of token words and token define names, creates the trie in memory, and dumps the trie to standard out as static C data arrays. This scheme facilitates adding or deleting a word from the trie.

Listing 3

```
/*****************************************************************************
* Module    :   Lexical Analyzer --- Header file containing token value
*               enumeration, type definitions and function prototypes for
*               the lexical analyzer functions.
*
*               Copyright (C) 1990 John W. M. Stevens, All Rights Reserved
*
* Author    :   John W. M. Stevens
*****************************************************************************/

#if     ! defined( LEXICAL_ANALYZER )
#define     LEXICAL_ANALYZER        1

#include    <dos.h>

#define     TRUE        1
#define     FALSE       0
#define     ERROR       -1
#define     OK          0

#define     PATH_SZ     65
typedef     unsigned    int     UINT;
typedef     unsigned    char    UCHAR;
typedef     char        PATH[PATH_SZ];

/* Definition of structure filled in and returned by lex.   */
typedef struct  {
    char    str[257];
    long    no;
    struct  time    ftime;
    struct  date    fdate;
} TOKEN;

/* Token defines.   */
enum    tkn_en  {
    STRING = 128,
    NUMBER,     TIME,       DATE,

    L_PAREN,    R_PAREN,    COMMA,      F_SLASH,    ACTION,
    AFTER,      AND,        ARCHIVE,    ATTRIBUTES, BEFORE,
    DIRECTORY_T, EXEC,      FILES,      HIDDEN,     LABEL,
    MODIFIED,   NAME,       NOT,        OR,         PRINT,
    READONLY,   RECURS,     SEARCH,     SELECT,     SYSTEM,
    L_BRACE,    BAR,        R_BRACE
};
typedef enum    tkn_en  TKNS;

/* Function prototypes. */
extern  TKNS    Lex(TOKEN *);
extern  void    OpenPrg(char *);
extern  void    ParsErr(char *);

#endif
/* End of File */
```

Example Lexical Analyzer

The example lexical analyzer uses a trie that contains the keywords and token values defined in Listing 2. The first word on the line is the keyword, and the second word is the token value enumeration label for that keyword. The trie creation program processes this file. The program output is captured in a file that will be included in the source code file for the lexical analyzer.

Listing 3 contains the token value enumeration, function prototypes, and type definitions necessary to use the lexical analyzer. To increase the readability of the parser source code, I've selected enumeration labels that are as similar as possible to the keywords they represent. Because of name space collision with type and/or define names used by the C compiler, some of the enumeration labels are postfixed with the string _T.

(text continued on page 83)

Listing 4

```
/******************************************************************************
* Module    :   Lexical Analyzer --- Process the input text file into tokens
*                   that the parser can understand.
*
*               Copyright (C) 1990 John W. M. Stevens, All Rights Reserved
*
* Routines  :   Lex     - Return the next token from the file.
*               OpenPrg - Open the source file.
*               ParsErr - Report a parsing error.
*
* Author    :   John W. M. Stevens
******************************************************************************/

#include    <stdio.h>
#include    <stdlib.h>
#include    <ctype.h>
#include    <string.h>

#include    "lex.h"

/* Structure of trie branch.    */
typedef struct  key_st  {
    char    c;
    TKNS    token;
    struct  key_st  *child;
} NODE;

/* Constants local to this file.    */
#define     MAX_STR     256
#define     NOT_FND     -2

/* Object Data. */
static  char    word[MAX_STR + 1];  /* Last string analyzed.              */
static  char    PrvWd[MAX_STR + 1]; /* Previous word.                    */
static  int     LnNo = 0;           /* The current line number in the file. */
static  FILE    *PrgFl;             /* File pointer.                     */
```

Listing 4 — Cont'd

```
/* Trie data structure containing all the keywords and punctuation
marks for
*   the language being tokenized.
*/
static
NODE    T5[2] = {
    {   ' ',            2, NULL     },
    {   'n',       ACTION, NULL     }
};

static
NODE    T4[2] = {
    {   ' ',            2, NULL     },
    {   'o',            0, T5   }
};

static
NODE    T3[2] = {
    {   ' ',            2, NULL     },
    {   'i',            0, T4   }
};

static
NODE    T2[2] = {
    {   ' ',            2, NULL     },
    {   't',            0, T3   }
};

static
NODE    T8[2] = {
    {   ' ',            2, NULL     },
    {   'r',        AFTER, NULL     }
};

static
NODE    T7[2] = {
    {   ' ',            2, NULL     },
    {   'e',            0, T8   }
};

static
NODE    T6[2] = {
    {   ' ',            2, NULL     },
    {   't',            0, T7   }
};

static
NODE    T9[2] = {
    {   ' ',            2, NULL     },
    {   'd',          AND, NULL     }
};

static
NODE    Te[2] = {
    {   ' ',            2, NULL     },
    {   'e',      ARCHIVE, NULL     }
};

static
NODE    Td[2] = {
    {   ' ',            2, NULL     },
    {   'v',            0, Te   }
};

static
NODE    Tc[2] = {
    {   ' ',            2, NULL     },
    {   'i',            0, Td   }
};

static
NODE    Tb[2] = {
    {   ' ',            2, NULL     },
    {   'h',            0, Tc   }
};

static
NODE    Ta[2] = {
    {   ' ',            2, NULL     },
    {   'c',            0, Tb   }
};

static
NODE    T16[2] = {
    {   ' ',            2, NULL     },
    {   's',   ATTRIBUTES, NULL     }
};

static
NODE    T15[2] = {
    {   ' ',            2, NULL     },
    {   'e',            0, T16 }
};

static
NODE    T14[2] = {
    {   ' ',            2, NULL     },
    {   't',            0, T15 }
};

static
NODE    T13[2] = {
    {   ' ',            2, NULL     },
    {   'u',            0, T14 }
};

static
NODE    T12[2] = {
    {   ' ',            2, NULL     },
    {   'b',            0, T13 }
};

static
NODE    T11[2] = {
    {   ' ',            2, NULL     },
    {   'i',            0, T12 }
};
```

Listing 4 — Cont'd

```
static
NODE    T10[2] = {
    {   ' ',            2, NULL    },
    {   'r',            0, T11 }
};

static
NODE    Tf[2] = {
    {   ' ',            2, NULL    },
    {   't',            0, T10 }
};

static
NODE    T1[6] = {
    {   ' ',            6, NULL    },
    {   'c',            0, T2  },
    {   'f',            0, T6  },
    {   'n',            0, T9  },
    {   'r',            0, Ta  },
    {   't',            0, Tf  }
};

static
NODE    T1b[2] = {
    {   ' ',            2, NULL    },
    {   'e',      BEFORE, NULL    }
};

static
NODE    T1a[2] = {
    {   ' ',            2, NULL    },
    {   'r',            0, T1b }
};

static
NODE    T19[2] = {
    {   ' ',            2, NULL    },
    {   'o',            0, T1a }
};

static
NODE    T18[2] = {
    {   ' ',            2, NULL    },
    {   'f',            0, T19 }
};

static
NODE    T17[2] = {
    {   ' ',            2, NULL    },
    {   'e',            0, T18 }
};

static
NODE    T23[2] = {
    {   ' ',            2, NULL    },
    {   'y',   DIRECTORY_T,    NULL    }
};

static
NODE    T22[2] = {
    {   ' ',            2, NULL    },
    {   'r',            0, T23 }
};

static
NODE    T21[2] = {
    {   ' ',            2, NULL    },
    {   'o',            0, T22 }
};

static
NODE    T20[2] = {
    {   ' ',            2, NULL    },
    {   't',            0, T21 }
};
```

Listing 4 — Cont'd

```
static
NODE    T1f[2] = {
        { ' ',          2, NULL    },
        { 'c',          0, T20 }
};

static
NODE    T1e[2] = {
        { ' ',          2, NULL    },
        { 'e',          0, T1f }
};

static
NODE    T1d[2] = {
        { ' ',          2, NULL    },
        { 'r',          0, T1e }
};

static
NODE    T1c[2] = {
        { ' ',          2, NULL    },
        { 'i',          0, T1d }
};

static
NODE    T26[2] = {
        { ' ',          2, NULL    },
        { 'c',       EXEC, NULL    }
};

static
NODE    T25[2] = {
        { ' ',          2, NULL    },
        { 'e',          0, T26 }
};

static
NODE    T24[2] = {
        { ' ',          2, NULL    },
        { 'x',          0, T25 }
};

static
NODE    T2a[2] = {
        { ' ',          2, NULL    },
        { 's',      FILES, NULL    }
};

static
NODE    T29[2] = {
        { ' ',          2, NULL    },
        { 'e',          0, T2a }
};

static
NODE    T28[2] = {
        { ' ',          2, NULL    },
        { 'l',          0, T29 }
};

static
NODE    T27[2] = {
        { ' ',          2, NULL    },
        { 'i',          0, T28 }
};

static
NODE    T2f[2] = {
        { ' ',          2, NULL    },
        { 'n',     HIDDEN, NULL    }
};

static
NODE    T2e[2] = {
        { ' ',          2, NULL    },
        { 'e',          0, T2f }
};

static
NODE    T2d[2] = {
        { ' ',          2, NULL    },
        { 'd',          0, T2e }
};

static
NODE    T2c[2] = {
        { ' ',          2, NULL    },
        { 'd',          0, T2d }
};

static
NODE    T2b[2] = {
        { ' ',          2, NULL    },
        { 'i',          0, T2c }
};

static
NODE    T33[2] = {
        { ' ',          2, NULL    },
        { 'l',      LABEL, NULL    }
};

static
NODE    T32[2] = {
        { ' ',          2, NULL    },
        { 'e',          0, T33 }
};

static
NODE    T31[2] = {
        { ' ',          2, NULL    },
        { 'b',          0, T32 }
};
```

Listing 4 — Cont'd

```
static
NODE    T30[2] = {
    {   ' ',            2, NULL   },
    {   'a',            0, T31 }
};

static
NODE    T3a[2] = {
    {   ' ',            2, NULL   },
    {   'd',     MODIFIED, NULL   }
};

static
NODE    T39[2] = {
    {   ' ',            2, NULL   },
    {   'e',            0, T3a }
};

static
NODE    T38[2] = {
    {   ' ',            2, NULL   },
    {   'i',            0, T39 }
};

static
NODE    T37[2] = {
    {   ' ',            2, NULL   },
    {   'f',            0, T38 }
};

static
NODE    T36[2] = {
    {   ' ',            2, NULL   },
    {   'i',            0, T37 }
};

static
NODE    T35[2] = {
    {   ' ',            2, NULL   },
    {   'd',            0, T36 }
};

static
NODE    T34[2] = {
    {   ' ',            2, NULL   },
    {   'o',            0, T35 }
};

static
NODE    T3d[2] = {
    {   ' ',            2, NULL   },
    {   'e',         NAME, NULL   }
};

static
NODE    T3c[2] = {
    {   ' ',            2, NULL   },
    {   'm',            0, T3d }
};

static
NODE    T3e[2] = {
    {   ' ',            2, NULL   },
    {   't',          NOT, NULL   }
};

static
NODE    T3b[3] = {
    {   ' ',            3, NULL   },
    {   'a',            0, T3c },
    {   'o',            0, T3e }
};

static
NODE    T3f[2] = {
    {   ' ',            2, NULL   },
    {   'r',           OR, NULL   }
};

static
NODE    T43[2] = {
    {   ' ',            2, NULL   },
    {   't',        PRINT, NULL   }
};

static
NODE    T42[2] = {
    {   ' ',            2, NULL   },
    {   'n',            0, T43 }
};

static
NODE    T41[2] = {
    {   ' ',            2, NULL   },
    {   'i',            0, T42 }
};

static
NODE    T40[2] = {
    {   ' ',            2, NULL   },
    {   'r',            0, T41 }
};

static
NODE    T4a[2] = {
    {   ' ',            2, NULL   },
    {   'y',     READONLY, NULL   }
};
```

Listing 4 — Cont'd

```
static
NODE    T49[2] = {
    {   ' ',          2, NULL    },
    {   'l',          0, T4a }
};

static
NODE    T48[2] = {
    {   ' ',          2, NULL    },
    {   'n',          0, T49 }
};

static
NODE    T47[2] = {
    {   ' ',          2, NULL    },
    {   'o',          0, T48 }
};

static
NODE    T46[2] = {
    {   ' ',          2, NULL    },
    {   'd',          0, T47 }
};

static
NODE    T4d[2] = {
    {   ' ',          2, NULL    },
    {   's',     RECURS, NULL    }
};

static
NODE    T4c[2] = {
    {   ' ',          2, NULL    },
    {   'r',          0, T4d }
};

static
NODE    T4b[2] = {
    {   ' ',          2, NULL    },
    {   'u',          0, T4c }
};

static
NODE    T45[3] = {
    {   ' ',          3, NULL    },
    {   'a',          0, T46 },
    {   'c',          0, T4b }
};

static
NODE    T44[2] = {
    {   ' ',          2, NULL    },
    {   'e',          0, T45 }
};

static
NODE    T52[2] = {
    {   ' ',          2, NULL    },
    {   'h',     SEARCH, NULL    }
};

static
NODE    T51[2] = {
    {   ' ',          2, NULL    },
    {   'c',          0, T52 }
};

static
NODE    T50[2] = {
    {   ' ',          2, NULL    },
    {   'r',          0, T51 }
};

static
NODE    T55[2] = {
    {   ' ',          2, NULL    },
    {   't',     SELECT, NULL    }
};

static
NODE    T54[2] = {
    {   ' ',          2, NULL    },
    {   'c',          0, T55 }
};

static
NODE    T53[2] = {
    {   ' ',          2, NULL    },
    {   'e',          0, T54 }
};

static
NODE    T4f[3] = {
    {   ' ',          3, NULL    },
    {   'a',          0, T50 },
    {   'l',          0, T53 }
};

static
NODE    T59[2] = {
    {   ' ',          2, NULL    },
    {   'm',     SYSTEM, NULL    }
};

static
NODE    T58[2] = {
    {   ' ',          2, NULL    },
    {   'e',          0, T59 }
};
```

Listing 4 — Cont'd

```
static
NODE    T57[2] = {
    {   ' ',            2, NULL    },
    {   't',            0, T58 }
};

static
NODE    T56[2] = {
    {   ' ',            2, NULL    },
    {   's',            0, T57 }
};

static
NODE    T4e[3] = {
    {   ' ',            3, NULL    },
    {   'e',            0, T4f },
    {   'y',            0, T56 }
};

static
NODE    T0[21] = {
    {   ' ',           21, NULL    },
    {   '(',      L_PAREN, NULL    },
    {   ')',      R_PAREN, NULL    },
    {   ',',        COMMA, NULL    },
    {   '/',      F_SLASH, NULL    },
    {   'a',            0, T1  },
    {   'b',            0, T17 },
    {   'd',            0, T1c },
    {   'e',            0, T24 },
    {   'f',            0, T27 },
    {   'h',            0, T2b },
    {   'l',            0, T30 },
    {   'm',            0, T34 },
    {   'n',            0, T3b },
    {   'o',            0, T3f },
    {   'p',            0, T40 },
    {   'r',            0, T44 },
    {   's',            0, T4e },
    {   '{',      L_BRACE, NULL    },
    {   '|',          BAR, NULL    },
    {   '}',      R_BRACE, NULL    }
};

/*---------------------------------------------------------
| Routine   :   OpenPrg() --- Open the ASCII text file
|                   that contains the back up program.
|
| Inputs    :   FileNm  - File name of source file.
---------------------------------------------------------*/

void    OpenPrg(char    *FileNm)
{
    /* Open the program script file.    */
    if ((PrgFl = fopen(FileNm, "rt")) == NULL)
    {
        fprintf(stderr, "OpenPrg (fopen) : Could not open
file '%s' for "
            "reading.\n", FileNm);
        exit( -1 );
    }

    /* Initialize object variables. */
    *word = *PrvWd = '\0';
}
```

Listing 4 — Cont'd

```
/*---------------------------------------------------------------------------
| Routine   :   ParsErr() --- Report a parse error.
|
| Inputs    :   Err - Error string.
---------------------------------------------------------------------------*/

void    ParsErr(char    *Err)
{
    /* Print line number and error message. */
    fprintf(stderr, "Error in Line: %d, %s.\n", LnNo + 1, Err);

    /* If there is a previous word, show it.    */
    if ( *word )
        fprintf(stderr, "\tOn or after word '%s'\n", word);
    exit( -1 );
}

/*---------------------------------------------------------------------------
| Routine   :   TrieSrch() --- Search the trie for a key word.
|
| Inputs    :   Trie    - The trie level pointer.
|               ch      - The current character to search for.
|               WordPtr - The pointer to the current byte of the word buffer.
| Returns   :   Returns either a token value or
|                   NOT_FND - For key word not found.
|                   EOF     - For end of file.
---------------------------------------------------------------------------*/

static
int     TrieSrch(NODE   *Trie,
                 int    ch,
                 char   *WordPtr)
{
    register    int     mid;        /* Mid point of array piece.            */
    register    TKNS    ret;        /* Return value of comparison.          */

    auto        int     lo;         /* Limits of current array piece for    */
    auto        int     hi;         /*      binary search.                  */

    /* Make sure that input is lower case.  */
    ch = tolower( ch );

    /* Search for a token.  */
    hi = Trie[0].token - 1;
    lo = 1;
    do
    {
        /* Find mid point of current array piece.   */
        mid = (lo + hi) >> 1;

        /* Do character comparison. */
        ret = ch - Trie[mid].c;

        /* Fix the array limits.    */
        if (ret <= 0)
            hi = mid - 1;
        if (ret >= 0)
            lo = mid + 1;

    }  while (hi >= lo);

    /* If the character matches one of the entries in this level and this
     *  entry has a child, recurse.  If a match is found but the matching
     *  entry has no child, return the token value associated with the
     *  match.  If the return value from the recursive call indicates that
     *  no match was found at a lower level, return the token value
     *  associated with the match at this level of the trie.
     */
    if (ret == 0)
    {
        /* Save the current character in the buffer for error reporting.    */
        *WordPtr++ = ch;
```

The C Users Order Form

	Quantity	Code #	Item	Unit Price	Total
C Users Journal			1-year subscription (12 issues)	$29.95 US, $54 CAN/MEX, $65 overseas	
			2-year subscription (24 issues)	$56 US, $89 CAN/MEX, $123 overseas	
			3-year subscription (36 issues)	$79 US, $122 CAN/MEX, $175 overseas	
			Magazine Code Disk (Mo: Yr:)	$5.00	
			Magazine Code Disks — 1991 (V.9)	$20.00	
			Magazine Code Disks — 1990 (V.8)	$20.00	
			CUJ Back Issue (Mo: Yr.)	$7.50, $11.00 overseas	

	Quantity	Code #	Title	Unit Price	Total
Books See below for overseas shipping charges.		CAT1	The CUG Directory - Vol. I	$10.00	
		CAT2	The CUG Directory - Vol. II	$10.00	
		CAT18	CUG Directory Volumes I & II	$18.00	

	Quantity	Code #	Disk Format/Size (MS-DOS, etc.)	# disks per volume	Unit Price	Total
Disks See below for overseas shipping charges. NOTE: Please indicate format and size of disks ordered.				( ) x $4.00		
				( ) x $4.00		
				( ) x $4.00		
				( ) x $4.00		

Please send more information about:

- ☐ SysAdmin
- ☐ The C Users Journal
- ☐ Windows™/DOS
- ☐ The C Users Bookstore
- ☐ The C Users' Group

All payments must be in US dollars (Mastercard/VISA accepted)
NOTE: Call 913-841-1631 for special shipping!

Name ______________________

Company ______________________

Address ______________________

____

City ______________________

State ____________ Zip Code ____________

Country ______________________

Daytime Phone ______________________

☐ I am a current subscriber

OVERSEAS SHIPPING CHARGES	BOOKS - Add 45% + $3.50 s&h	
	DISKS - Add 30% + $3.50 s&h	
shipping charge in North America ($3.50)		

TOTAL	

☐ MC ☐ Visa Expiration Date ____________

Card # ______________________

Signature ______________________

Mail to: The C Users Journal, 1601 W. 23rd St., Suite 200
P.O. Box 3127, Lawrence KS 66046-0127

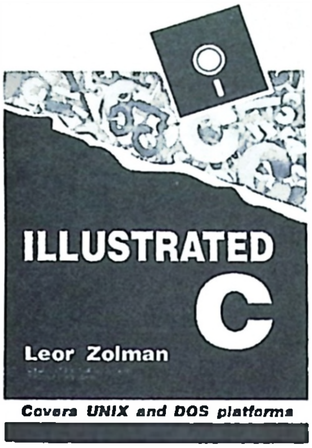
— For fastest service, call (913) 841-1631 —

C Users Bookstore

The Resource Center for C Programming Books

Hard-to-find titles and titles you won't find anywhere else

C++ • UNIX • ANSI/Standard C • Turbo C • Windows • X Window



Illustrated C

By Leor Zolman

Illustrated C explores the construction of several different applications, from start to finish. Through a focus on building useful, practical programs, this book gives the C programmer the "why and how" of application design and development. Each program is exhaustively annotated in a clear, readable style.

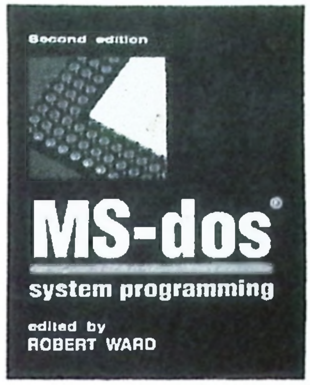
Illustrated C is a tutorial for the novice to intermediate programmer and a toolbox for all C programmers.

R&D Publications, 1992, 320 pp.

ISBN 0-923667-21-0

W34 .... $29.95

W36 w/ disk $39.95



MS-DOS System Programming / 2nd edition

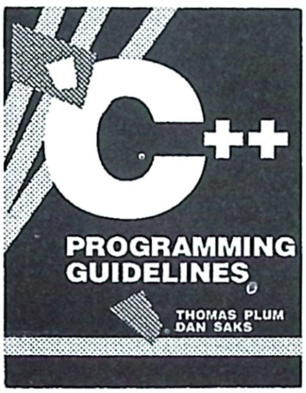
Edited by Robert Ward

This highly technical, focused work is written by working programmers for working programmers and designed to save the reader hours of work. How-to topics include: critical error handling, customizing the DOS boot strap, interrupt-driven I/O, manipulating environmental variables, event timing, writing TSRs and device drivers, and much more. The final chapter is an annotated bibliography of books designed to help the serious programmer develop a personal library of professional MS-DOS literature.

R&D Publications, 1991, 240 pp.

ISBN 0-923667-20-2

Z82 .... $24.95



C++ Programming Guidelines

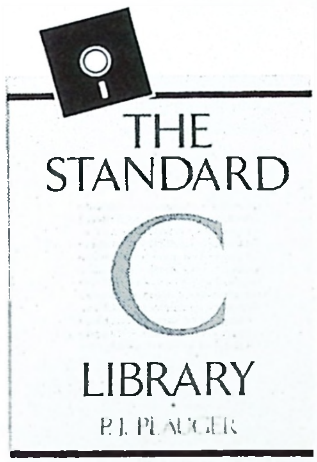
By Thomas Plum and Dan Saks

C++ Programming Guidelines gives professional C++ developers useful information for writing portable and maintainable C++ programs. Written for the professional C and C++ programmer, this book aids the production of efficient, portable programs which can be understood and maintained by other developers. Users learn both lexical layout and more fundamental semantics for data and variables, operators, control statements, and functions. C++ Programming Guidelines is arranged in a "manual-page" format for easy reference.

Plum-Hall, 1992, 265 pp.

ISBN 0-911537-10-4

W33 .... $34.95



The Standard C Library

By P.J. Plauger

The Standard C Library shows you how to use all of the library functions mandated by the ANSI and ISO Standards for the programming language C. To help you understand how to use the library, this book also shows you how to implement it. Approximately 9,000 lines of tested, working code that is highly portable across diverse computer architectures is included.

Prentice Hall, 1991, 498 pp.

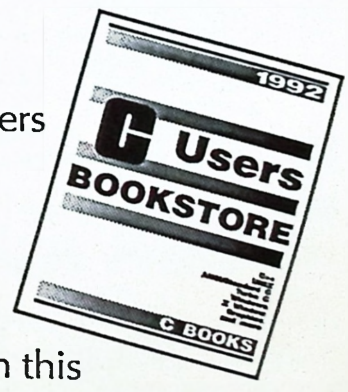
ISBN 0-13-131509-9

Z81 .... $28.00

DK1 w/ disk $77.95

Use the order form on the opposite page to order these titles or any of 150 books in the Bookstore.

FREE 1992 C Users Bookstore Guide



Get your complete listing and description of all the books in the C Users Bookstore. The 24-page reference guide is yours for the asking. Titles are arranged by subject for easy reference and are indexed by title and author.

CALL TODAY—(913) 841-1631. Or, use the Reader Service card in this magazine. Request 153 on the Reader Service Card.

The C Users' Group

The C Users' Group collects and maintains the public domain C source code and shareware listed below. Distribution fee is $4.00 per disk. There is a $3.50 shipping and handling charge per order U.S.; 30% overseas. If there are multiple disks in a volume, that number is indicated in parentheses. Due to updates, the number of disks per volume is subject to change without notice. The CUG Library supports code written for MS-DOS, CP/M and UNIX. Volumes are continually added.

For volumes 101-156 see the CUG Library Directory.

157 — QE for MSDOS
158 — QE for BDS C
159 — Adventure, MSDOS
160 — Programs from Learning to Program in C (Plum Hall)
161 — Programs from Efficient C (Plum Hall)
162 — Mchip 80
163 — Small C for MSDOS (PCC, Toronto) (2)
164 — Windows
165 — Programs from Reliable Data Structures in C (Plum Hall)
166 — CUG Directory (#101-#155)
167 — Windows, Unix-like Util
168 — Simple Database (PC-SIG 147)
169 — Miscellany IV (PC-SIG 314)
170 — Miscellany V (PC-SIG 315)
171 — Miscellany VI (PC-SIG 341)
172 — LEX Part 1
173 — LEX Part 2
176 — Xlisp v 1.6 (2)
177 — Dr. Shaw's DOS Shell
178 — TVX Source
179 — TVX Doc & Exec.
181 — CFORUM & Lzw
182 — UNIX BBS
183 — PC Tools II
184 — RunAMD
185 — Sorts
186 — Make & Aim
187 — Functions IV
188 — Utilities VI
189 — PC Tools III
190 — Steve Passe's 68K Assembler
191 — Miscellany VII
192 — Miscellany VIII
193 — Crypto Toolbox
194 — JUGPDS 17
195 — JUGPDS 18
196 — JUGPDS 19
197 — MicroEmacs v 3.9 Exec. & Doc.
198 — MicroEmacs v 3.9 Source (2)
199 — GED MSDOS text editor
200 — Small c Interpreter
201 — MSDOS System Support
202 — KAREL for MSDOS
203 — YAM for MSDOS
204 — 68000 C Compiler
205 — Utilities VII
206 — Checkbook Register Doc & Exec
207 — Checkbook Register Doc & Source
208 — e for CP/M 68K
209 — Simplex Curve Fitting
210 — Simulation and Loadable BIOS for CP/M
211 — Search, Sort and Merge
212 — Simulation for BDS
213 — Van Nuys Enhancements
214 — File Display Util
215 — BBS for BDS C
216 — Zmodem & Saveram (2)
217 — Spell & Dict Part I
218 — Dict Part II (2)
219 — 6502 Cross Assemblers
220 — Windows BOSS (3)
221 — 6809 C for flex
222 — Small C for CP/M Doc & Exec
223 — Small C for CP/M Source (2)
224 — Utilix
225 — Utilities and Zmath
226 — ART-CEE
227 — Portable Graphics (2)
228 — Miscellany IX
229 — Little Smalltalk —UNIX Part 1
230 — Little Smalltalk —UNIX Part 2
231 — Little Smalltalk —Unbundled Part 1
232 — Little Smalltalk —Unbundled Part 2
233 — Make And Other Utilities
234 — XDIR File and Maintenance Utilities
235 — Overview File and Maintenance Utilities (2)
236 — Highly Portable Utilities (CUG Starter Disk)
237 — GRAD Graphics - main library
238 — GRAD Graphics - demos
239 — P.C. Gammon For MSC4
240 — P.C. Gammon For Turbo
241 — Inference Engine & Rule Based Compiler
242 — Still More Cross Assemblers
243 — DECUS C Preprocessor
244 — deBruijn
245 — Linear Cellular Automata
246 — Cycles, Mandelbrot Graphics (2)
247 — Miracl (3)
248 — Micro Spell
249 — C Spot Run Library (2)
250 — 68K FP Library & Mandelbrot Graphics
251 — Pull Down Menus & Adventure System
252 — C Tutor Doc
253 — C Tutor Source
255 — EGA Graphics Library
256 — DeSmet Carry Routines (2)
257 — C Tutor doc (Turbo C)
258 — C Tutor Source (Turbo C)
259 — Console I/O & Withers Tools
260 — Zmodem, CU, tty library
261 — 68k Cross Assembler MS-DOS
262 — Ramey Tools
263 — C_wndw Toolkit (2)
264 — NRO & Other PC Tools
265 — cpio Installation Kit † Call CUG for ordering Instructions (2)
266 — Micro PLOX
267 — 8085, 2650 & 2656 Cross Assemblers
268 — Unicorn Library Turbo C (2)
269 — Unicorn Library MSC
270 — Miscellany X
271 — Steven's Library Turbo C (2)
272 — Steven's Library Datalight C
273 — Turbo C Utilities
274 — Arrays for C
275 — Linear Cellular Automata II (2)
276 — Z80 and 6804 cross assemblers
277 — HP Plotter Library and XMODEM
278 — CXL Function Library (3)
279 — CROBOTS
280 — Software Tools Souce
281 — Unicorn Library v5.1 (2)
282 — Quip and Graphics
283 — FAFNIR (3)
284 — Portable 8080 Emulator
285 — Bison for MS-DOS (3)
286 — GRAD for MSC (2)
287 — GRAD for Turbo C (3)
288 — Traveling Salesman, SD and Master Environment Access
289 — Othello
290 — FLEX (2)
291 — JJB for Quick C & Turbo C
292 — ASxxxx C Cross Assembers (4)
293 — 3D Med. Imaging, Source (2)
294 — 3D Med. Imaging, sample images (4)
296 — C to C++ Migrator
297 — Small Prolog
298 — PC Curses
299 — MEL and BP
300 — MAT_LIB
301 — BGI Applications (2)
302 — 3-D Transforms
303 — MC68K Disassembler
304 — ROFF5
305—HGA Mandelbrot Explorer /Card Games (2)
306 — Threads and Synapsys
307 — ADU & COMX (Device Driver)
308 — MSU, REMZ & LIST (2)
309 — 6809 C Compiler for MS-DOS
310 — Little Smalltalk for MS-DOS (2)
311 — DB package
312 — Make-Maker
313 — STEVIE
314 — MNP C Library
315 — FTGRAPH
316 — AS8 Cross Assembler
317 — Group3 Image Processing
318 — RED (2)
319 — CPP (2)
320 — Convolution Image Process
321 — Mouse Trap Library
322 — Doctor's Tools
323 — Fireworks and Adventure
324 — WGCONIO
325 — VGA Graphics Library (2)
326 — SoftC Database Library (2)
327 — Panels for C
328 — WTWG (2)
329 — UNIX Tools for PC (2)
330 — CTask (3)
331 — SE Editor
332 — PCcurses (2)
333 — gAWK (2)
334 — GNUPLOT (3)
335 — Frankenstein Cross Assemblers (4)
336 — EGAPAL/EDIPAL
337 — Designing Screen Interfaces in C
338 — 68K C Compiler and Assembler (2)
339 — CTRLCLIB (2)
340 — C-Window
341 — Orbit Propagation
342 — I8255 Interface Library
343 — C Image Processing System (4)
344 — Grab Bag #1 (2)
345 — TLC/TLP
346 — ASxxxx Cross Assembler Part 2 (2)
347 — TAVL Tree
348 — 8048 Disassembler/Z80 Assembler
349 — Simulation Subroutine Set
350 — PCX Graphics Library
351 — UltraWin
352 — String and Vlist
353 — C++ Tutor (2)
354 — CES Mouse Tools Library w/ Joystick
355 — Sherlock for MS-DOS (3)
356 — Sherlock for Macintosh (3)
357 — CSTAR (2)
358 — cbase
359 — GNU C/C++ for 386 (12)
360 — Uspell (2)
361 — Gadgets and Term

Be sure to specify format when ordering disk. We can supply these volumes in almost any MS-DOS format ($3\frac{1}{2}$" or $5\frac{1}{4}$"), UNIX tar or cpio archives ($5\frac{1}{4}$" only), or Macintosh.

The C Users' Group, 1601 W. 23rd St., Suite 200, P.O. Box 3127, Lawrence, KS 66046-0127
(913) 841-1631 • FAX (913) 841-2624

(text continued from page 71)

Listing 4 contains the lexical analyzer. I've extracted this code from a program that acts as a user-programmable file selection shell. The function *OpenPrg()* initializes the lexical analyzer by opening the file that contains the source code to be analyzed. The parser then calls the function *Lex()* repeatedly to get tokens. Each time *Lex()* is called, it begins by reading and throwing away both white space characters (space, tab and newline characters) and comments. When the first character of a suspected keyword is found, it breaks out of the loop and attempts to get either a string constant, integer constant, time, or date.

If the input is not a constant of some type, the trie search function is called. The function *TrieSrch()* begins by attempting to find the input character in the trie node. *TrieSrch()* accepts a pointer to a node of a trie, a character to search for, and a pointer to a buffer for storing the word read from the input file. The function uses a binary search because the characters in a trie node are stored in sorted order.

If the input character is found in the trie node, *TrieSrch()* saves it in the word buffer. If the matching character in the trie node has a pointer to a child trie node, *TrieSrch()* reads another character from the file and calls itself recursively. If the return value from the recursive call indicates that the character was not found, *TrieSrch()* assumes that the input character for this call was the last character of a legal word and unreads the character read for the recursive call. The token value of the input character to this call is returned.

If the matching character does not have a pointer to a child trie node, the keyword buffer is *NUL*-terminated and the token value stored with the matching character is returned. If the input character is not found in the trie node, *TrieSrch()* *NUL*-terminates the keyword buffer and returns a value indicating that the character was not found.

Figure 2 presents an algorithm in structured English for separating words from an input character stream and searching for them in a search trie.

(text continued on page 85)

Listing 4 — Cont'd

```
/* Are we looking for more characters in this string?   */
if ( Trie[mid].child )
{
    /* Get the next character.  */
    if ((ch = fgetc( PrgFl )) == EOF)
        return( EOF );

    /* Search next level.   */
    if ((ret = TrieSrch(Trie[mid].child, ch, WordPtr)) == NOT_FND)
    {
        ungetc(ch, PrgFl);
        return( Trie[mid].token );
    }
    return( ret );
}
else
```

Listing 4 — Cont'd

```
            {
                /* Properly NUL terminate the buffer that the keyword is
                *   being saved in and return the token value.
                */
                *WordPtr = '\0';
                return( Trie[mid].token );
            }
        }

        /* Terminate string in keyword buffer and return not found. */
        *WordPtr = '\0';
        return( NOT_FND );
}

/*-----------------------------------------------------------------------------
| Routine   :   GetNo --- Get a number from the file.
|
| Inputs    :   word     - Pointer to word buffer for error reporting.
| Outputs   :   RetNo    - Returns the number read from the file.
|
| Returns   :   Returns the last character read from the file or EOF.
-----------------------------------------------------------------------------*/

static
int     GetNo(char  **word,
              long  *RetNo)
{
    auto        int     c;

    /* Get number.  */
    *RetNo = 0L;
    while ((c = fgetc( PrgFl )) >= '0' && c <= '9')
    {
        /* Save character in word buffer.   */
        *(*word)++ = c;

        /* Calculate value of number.   */
        *RetNo = *RetNo * 10L + (long) (c - '0');
    }
    return( c );
}

/*-----------------------------------------------------------------------------
| Routine   :   Lex() --- Get the next key word from the input file.
|
| Outputs   :   sym - The symbolic data read from the file.
|
| Returns   :   Returns the token read or EOF.
-----------------------------------------------------------------------------*/

TKNS    Lex(TOKEN   *sym)
{
    register    int     i;
    register    int     tkn;
    auto        int     ch;
    auto        char    *bf;

    /* Strip comments and white space.  If the character read is a '#',
    *   every thing to the end of the line is a comment.
    */
    ch = fgetc( PrgFl );
    while (ch == ' ' || ch == '\t' || ch == '\n' || ch == '#')
    {
        /* Process the special characters '#' and '\n'.     */
        if (ch == '\n')
            /* End of line, increment the line number.  */
            LnNo++;
        else if (ch == '#')
        {
            /* Found a comment character, strip all characters to end
            *   of line and increment the line number.
            */
            while (fgetc( PrgFl ) != '\n')
                ;
            LnNo++;
        }
```

(text continued from page 83)

Summary

A trie is probably not the most efficient data structure for determining the legality of an input word. A sorted table of strings searched with a binary search would probably be faster and more memory efficient.

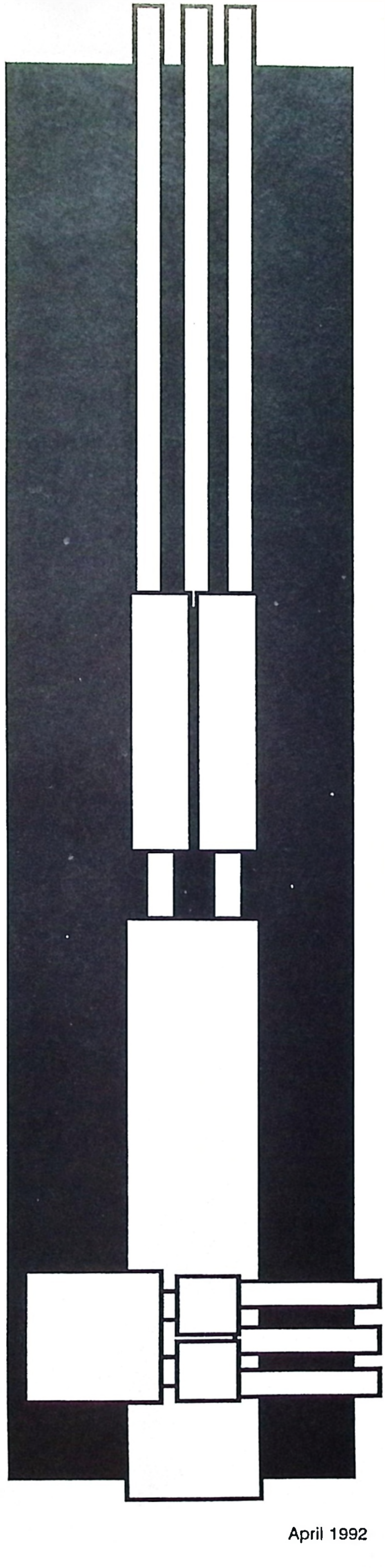
So why use a trie if it isn't as fast or efficient as other methods? Since the hardest part of writing a lexical analyzer is in breaking an undifferentiated stream of input characters into words, the beauty of a trie is that it groups characters into words and determines their legality at the same time. It is also, in my opinion, a more elegant solution. This alone is reason enough for me to use a trie. □

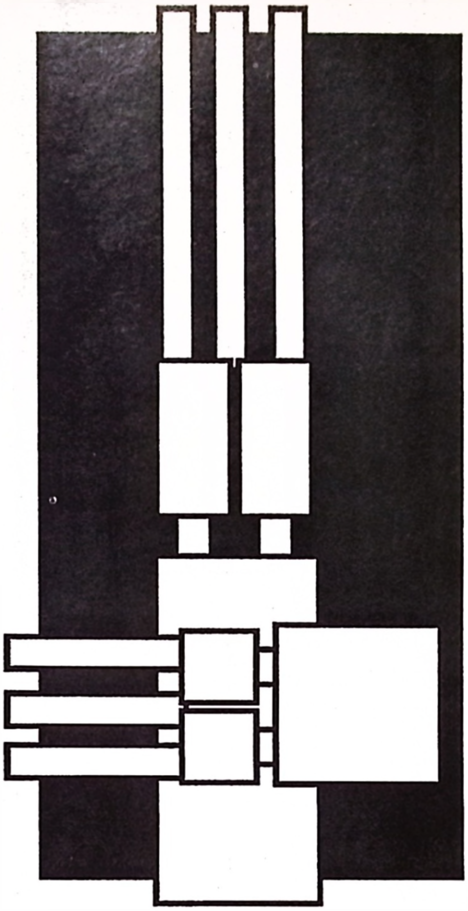
Listing 4 — Cont'd

```
        /* Get the next character.  */
        ch = fgetc( PrgFl );
    }

    /* Get strings, etc.    */
    if (ch == '"')
    {
        /* Get contents of string.  */
        bf = sym->str;
        for (i = 0; i < MAX_STR; i++)
            if ((ch = fgetc( PrgFl )) != '"' && ch != EOF)
                *bf++ = ch;
            else
                break;
        *bf = '\0';

        /* Return string token. */
        strcpy(word, sym->str);
        return( STRING );
    }
    else if (ch >= '0' && ch <= '9')
    {
        auto        long        no;

        /* Establish a pointer to the word buffer and unget the
        *   numeric character for re-reading.
        */
        bf = word;
        ungetc(ch, PrgFl);

        /* Get number, time or date.    */
        if ((ch = GetNo(&bf, &no)) == ':')
        {
            /* Getting time, not number.    */
            *bf++ = ch;
```

Listing 4 — Cont'd

```
        sym->ftime.ti_hour = (unsigned char) no;
        sym->ftime.ti_hund = (unsigned char) 0;

        /* Get minutes.     */
        if ((ch = GetNo(&bf, &no)) == ':')
        {
            /* Save minutes.    */
            *bf++ = ch;
            sym->ftime.ti_min = (unsigned char) no;

            /* Get seconds. */
            if ((ch = GetNo(&bf, &no)) == '.')
            {
                *bf = '\0';
                ParsErr( "Hundredths of seconds not allowed in "
                    "time expressions" );
            }
            sym->ftime.ti_sec = (unsigned char) no;
        }
        else
        {
            /* No seconds to get.   */
            sym->ftime.ti_min = (unsigned char) no;
            sym->ftime.ti_sec  = (unsigned char) 0;
        }

        /* This is a time.  */
        tkn = TIME;
    }
    else if (ch == '/')
    {
        /* Getting date, not number.    */
        *bf++ = ch;
        sym->fdate.da_mon = (char) no;
```

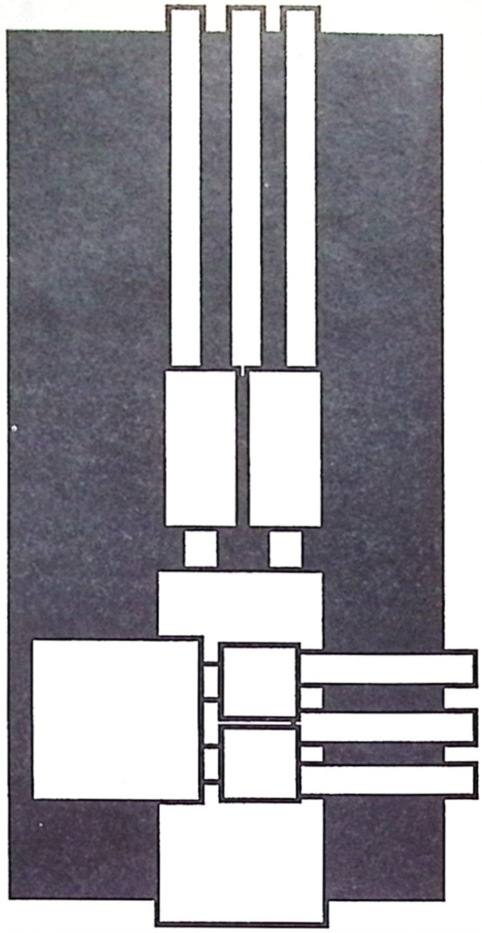


Listing 4 — Cont'd

```
            /* Get day.     */
            if ((ch = GetNo(&bf, &no)) == '/')
            {
                /* Save character.  */
                *bf++ = ch;
                sym->fdate.da_day = (char) no;

                /* Get year.    */
                ch = GetNo(&bf, &no);
                if (no > 1980L)
                    no -= 1980L;
                else if (no > 80L && no < 100L)
                    no -= 80L;
                else
                {
                    *bf = '\0';
                    ParsErr( "Error, bad year value in date expression." );
                }
                sym->fdate.da_year = (int) no;
            }
            else
            {
                *bf = '\0';
                ParsErr( "Missing year in date expression" );
            }

            /* This is a date.  */
            tkn = DATE;
        }
        else
        {
            /* Just an integer constant.    */
            sym->no = no;
            tkn = NUMBER;
        }
```

Listing 4 — Cont'd

```
        /* Return the unused character. */
        *bf = '\0';
        ungetc(ch, PrgFl);
        return( tkn );
    }
    else if (ch == EOF)
        return( EOF );

    /* Call the trie search routine to return the next token, EOF
     *  or NOT_FND.  If not found, print an error and quit.
     */
    if ((tkn = TrieSrch(TO, ch, word)) == NOT_FND)
    {
        /* Illegal first character in word. */
        if ( *PrvWd )
            fprintf(stderr, "Parse - Error in Line: %d, cannot identify "
                "word after '%s'.\n", LnNo + 1, PrvWd);
        else
            fprintf(stderr, "Parse - Error in Line: %d, cannot identify "
                "first word in file.\n", LnNo + 1);
        exit( -1 );
    }
    else if (tkn == 0)
    {
        /* Illegal word.    */
        fprintf(stderr, "Parse - Error in Line: %d, cannot identify "
            "word '%s'.\n", LnNo + 1, word);
        exit( -1 );
    }
    strcpy(PrvWd, word);

    /* Return the token found.  */
    return( tkn );
}

/* End of File */
```



Rex Jaeschke

Data Structures

Part 11: Yet More on Stacks

Handling Multiple Stacks of the Same Type

Last month's column described how to implement a stack and to use *push* and *pop* functions on it. However, it would be a waste to have a different set of *push* and *pop* functions for each stack. This month I will discuss how to share the code.

Consider Listing 1. An object of type *stack* contains the current context of a given stack. This context includes the stack's name (for debugging or trace-back purposes), the base address of the stack, the stack's size and its current stack pointer. The stack descriptors *stack1*, *stack2*, and *stack3* therefore describe the three different *int* stacks. The first stack is stored in a global array, the second in a file scope static array, while the third is allocated at runtime using *malloc*. In short, the stack management functions don't know and don't care where the stacks reside.

Listing 2 tells *push* and *pop* which stack to use but the notation is not particularly unwieldy. Listing 3 could be made a little bit cleaner by passing the stack descriptor by value, but that could be just a little more expensive since the descriptor is a structure. The output produced is shown in Figure 1.

Handling Multiple Stacks of Different Types

Can this idea be extended to manage stacks of different types? The answer is a qualified *yes*. One way could be to use generic pointers, as shown in Listing 4. Here, a *void* pointer is used to hold the stack's base address. This also requires an extra member to indicate the type of elements in any given stack, as in

```
void push(stack *, void *);
void *pop(stack *);
```

The interface to *push* and *pop* gets very messy, however. Since an object of arbitrary type cannot be passed by value, it must be passed by assigning it to a named object and then passing that object by address, as shown in Listing 5. (This eliminates the ability to push a constant directly.) This is possible since all data pointer types are compatible with *void **. Similarly, an arbitrary typed value can't be returned, so the address is returned instead.

To use the value returned by *pop* you must use an explicit cast as well as a dereference since *pop* returns a pointer (see Listing 6). In fact, since *pop* returns the address of the object just popped from the stack, if you don't dereference the pointer returned immediately, the location it points to will be overwritten by the very next push and the value popped will change.

Rex Jaeschke is an independent computer consultant, author, and seminar leader. He participates in both ANSI and ISO C Standards meetings and is the editor of The Journal of C Language Translation*, a quarterly publication aimed at implementors of C language translation tools. Readers are encouraged to submit column topics and suggestions to Rex at 2051 Swans Neck Way, Reston, VA 22091 or via UUCP at* rex@aussie.com.

Listing 1

```
#include <stdio.h>
#include <stdlib.h>

#define NUMELEMENTS(x) (sizeof(x)/sizeof(x[0]))

typedef struct {
        const char *stack_name;
        int *pstack;
        size_t stack_ptr;
        size_t max_stack;
} stack;

int array1[1];
static int array2[30];

stack stack1 = {"stack1", array1, 0, NUMELEMENTS(array1)};
stack stack2 = {"stack2", array2, 0, NUMELEMENTS(array2)};
stack stack3 = {"stack3", NULL,   0, 0};

/* End of File */
```

Listing 2

```
void push(stack *, int);
int pop(stack *);

main()
{
    int size = 50;

    stack3.pstack = malloc(size * sizeof(int));
    if (stack3.pstack == NULL) {
        printf("Can't allocate space for stack3\n");
        exit(1);
    }

    stack3.max_stack = size;

    push(&stack1, 10);
    push(&stack1, 12);
    push(&stack2, 15);
    push(&stack3, 20);
    printf("stk1: %d\n", pop(&stack1));
    printf("stk2: %d\n", pop(&stack2));
    printf("stk3: %d\n", pop(&stack3));
    printf("stk3: %d\n", pop(&stack3));

    return 0;
}

/* End of File */
```

Listing 3

```
void push(stack *st, int value)
{
    if (st->stack_ptr == st->max_stack)
        printf("Stack %s is full\n", st->stack_name);
    else
        st->pstack[st->stack_ptr++] = value;
}

int pop(stack *st)
{
    if (st->stack_ptr == 0) {
        printf("Stack %s is empty\n", st->stack_name);
        return 0;
    }

    return st->pstack[--st->stack_ptr];
}

/* End of File */
```

The source for *push* and *pop* is far from pretty. Since it cannot perform arithmetic (via subscripting) on *void* pointers, you must explicitly provide the scaling factor.

Using Unions

Another way to look at the problem is rather than have stacks of different types, have one stack that can handle objects of different types. You can achieve this via a union, as shown in Listing 7.

Each entry is a union of all the possible types along with a flag member that indicates which type this entry currently represents. We push nodes by value and return them by value using simple and obvious notation. If you try to pop from an empty stack, instead of complaining, *pop* returns a copy of a local node that has a special type field value of *Error*.

Next, consider Listing 8. Note the interesting controlling expression in the *while* loop — it uses the comma operator in an effective manner.

The stack management functions presented in Listing 9 are quite straightforward.

Opposing Stacks

A stack grows and shrinks from one end only so it is possible to have two stacks based at opposite ends of an array with their tops growing toward each other. This can save space if both stacks don't grow very large at the same time. However, when one is smaller, the other can grow larger and vice-versa.

The *dump_stack* function in Listing 10 allows us to see the contents of the underlying array as the two stacks grow and shrink. Note that it pops in a different order than it pushes, so the operations are staggered.

Figure 1

```
Stack stack1 is full
stk1: 10
stk2: 15
stk3: 20
Stack stack3 is empty
stk3: 0
```

Listing 5

```
main()
{
    int size = 50;
    int vali = 10;
    long vall = 456789;
    double vald = 123.456;

    stack3.pstack = malloc(size * sizeof(double));
    if (stack3.pstack == NULL) {
        printf("Can't allocate space for stack3\n");
        exit(1);
    }

    stack3.max_stack = size;

    push(&stack1, &vali);
    push(&stack2, &vall);
    push(&stack3, &vald);
    printf("stk1: %d\n", *((int *)pop(&stack1)));
    printf("stk2: %ld\n", *((long *)pop(&stack2)));
    printf("stk3: %f\n", *((double *)pop(&stack3)));

    return 0;
}

/* End of File */
```

Listing 4

```
#include <stdio.h>
#include <stdlib.h>

#define NUMELEMENTS(x) (sizeof(x)/sizeof(x[0]))
#define INT 1
#define LONG 2
#define DOUBLE 3

typedef struct {
        const char *stack_name;
        void *pstack;
        const int type;    /* type of entries */
        size_t stack_ptr;
        size_t max_stack;
} stack;

int array1[10];
static long array2[30];

stack stack1 = {"stack1", array1, INT,    0, NUMELEMENTS(array1)};
stack stack2 = {"stack2", array2, LONG,   0, NUMELEMENTS(array2)};
stack stack3 = {"stack3", NULL,   DOUBLE, 0, 0};

/* End of File */
```

Listing 6

```
void push(stack *st, void *pvalue)
{
        if (st->stack_ptr == st->max_stack)
                printf("Stack %s is full\n", st->stack_name);
        else
                switch (st->type) {

                case INT:
                        ((int *)st->pstack)[st->stack_ptr++] = *(int *)pvalue;
                        break;

                case LONG:
                        ((long *)st->pstack)[st->stack_ptr++] = *(long *)pvalue;
                        break;

                case DOUBLE:
                        ((double *)st->pstack)[st->stack_ptr++] = *(double *)pvalue;
                        break;
                }
}

void *pop(stack *st)
{
        if (st->stack_ptr == 0) {
                printf("Stack %s is empty\n", st->stack_name);
                return 0;
        }

        switch (st->type) {

        case INT:
                return &((int *)st->pstack)[--st->stack_ptr];
                break;

        case LONG:
                return &((long *)st->pstack)[--st->stack_ptr];
                break;

        case DOUBLE:
                return &((long *)st->pstack)[--st->stack_ptr];
                break;
        }
}

/* End of File */
```

Listing 7

```
#include <stdio.h>

enum node_type {Error, Char, Int, Double, String};

typedef struct {
        enum node_type type;
        union {
                char c;
                int i;
                double d;
                char *s;
        } value;
} node;

void push(node);
node pop(void);

/* End of File */
```

The stack in Listing 11 can only handle four elements. This helps you monitor the progress as elements are pushed and popped. Clearly, it would be larger in a real application. Of course, the bases of the two stacks are at either end of the underlying array and in one stack the stack pointer increments as we push and in the other it decrements. Stack overflow is detected when the two stack pointers bump into each other. Note that it's OK if both of them point to the same element since that last free element is available to which ever stack can use it first.

The two versions of *push* and *pop* in Listing 12 are different enough that it is not obvious you can write a single version

Listing 8

```
main()
{
    node n;

    n.value.d = 9.87;
    n.type = Double;
    push(n);

    n.value.s = "some text";
    n.type = String;
    push(n);

    n.value.i = 123;
    n.type = Int;
    push(n);

    n.value.c = 'A';
    n.type = Char;
    push(n);

    while (n = pop(), n.type != Error) {

        switch (n.type) {

        case Char:
            printf("Char = %c\n", n.value.c);
            break;

        case Int:
            printf("Int = %d\n", n.value.i);
            break;

        case Double:
            printf("Double = %f\n", n.value.d);
            break;

        case String:
            printf("String = %s\n", n.value.s);
            break;
        }
    }

    return 0;
}

/* End of File */
```

Figure 2

```
Stack contains:   0   0   0   0        sp1 = 0, sp2 = 3
Stack contains:  10   0   0   0        sp1 = 1, sp2 = 3
Stack contains:  10   0   0  12        sp1 = 1, sp2 = 2
Stack contains:  10  15   0  12        sp1 = 2, sp2 = 2
Stack contains:  10  15  34  12        sp1 = 2, sp2 = 1
Stack 1 is full
Stack contains:  10  15  34  12        sp1 = 2, sp2 = 1
Stack 2 is full
Stack contains:  10  15  34  12        sp1 = 2, sp2 = 1
stack 1: 15
Stack contains:  10  15  34  12        sp1 = 1, sp2 = 1
stack 1: 10
Stack contains:  10  15  34  12        sp1 = 0, sp2 = 1
Stack 1 is empty
stack 1: 0
Stack contains:  10  15  34  12        sp1 = 0, sp2 = 1
stack 2: 34
Stack contains:  10  15  34  12        sp1 = 0, sp2 = 2
stack 2: 12
Stack contains:  10  15  34  12        sp1 = 0, sp2 = 3
Stack 2 is empty
stack 2: 0
Stack contains:  10  15  34  12        sp1 = 0, sp2 = 3
```

Listing 9

```
#include <stdio.h>

#define STACK_SIZE 10

static node stack[STACK_SIZE];
static size_t stack_ptr = 0;

void push(node n)
{
    if (stack_ptr == STACK_SIZE)
        printf("Stack is full\n");
    else
        stack[stack_ptr++] = n;
}

node pop(void)
{
    static node error_node = {Error, 0};

    if (stack_ptr == 0)
        return error_node;
    else
        return stack[-stack_ptr];
}

/* End of File */
```

Listing 10

```
#include <stdio.h>

void push1(int);
void push2(int);
int pop1(void);
int pop2(void);
void dump_stack(void);

main()
{
        dump_stack(); push1(10);
        dump_stack(); push2(12);
        dump_stack(); push1(15);
        dump_stack(); push2(34);
        dump_stack(); push1(99);
        dump_stack(); push2(65);
        dump_stack(); printf("stack 1: %d\n", pop1());
        dump_stack(); printf("stack 1: %d\n", pop1());
        dump_stack(); printf("stack 1: %d\n", pop1());
        dump_stack(); printf("stack 2: %d\n", pop2());
        dump_stack(); printf("stack 2: %d\n", pop2());
        dump_stack(); printf("stack 2: %d\n", pop2());
        dump_stack();

        return 0;
}

/* End of File */
```

that is both elegant and efficient. The output produced is shown in Figure 2.

Note that even after a value is popped from a stack it still remains there — only the stack pointer is adjusted. This is exactly what happens when a C function returns; the values of

it's automatic variables are still on the stack but are outside the bounds of the newly adjusted stack pointer. They remain intact until that part of the stack is overwritten for some other purpose. □

Listing 11

```
#include <stdio.h>

#define STACK_SIZE 4

static int stack[STACK_SIZE];

static size_t stack_ptr1 = 0;
static size_t stack_ptr2 = STACK_SIZE - 1;

/* End of File */
```

Listing 12

```
void push1(int value)
{
        if (stack_ptr1 > stack_ptr2)
                printf("Stack 1 is full\n");
        else
                stack[stack_ptr1++] = value;
}

void push2(int value)
{
        if (stack_ptr1 > stack_ptr2)
                printf("Stack 2 is full\n");
        else
                stack[stack_ptr2--] = value;
}

int pop1(void)
{
        if (stack_ptr1 == 0) {
                printf("Stack 1 is empty\n");
                return 0;
        }

        return stack[--stack_ptr1];
}

int pop2(void)
{
        if (stack_ptr2 == STACK_SIZE - 1) {
                printf("Stack 2 is empty\n");
                return 0;
        }

        return stack[++stack_ptr2];
}

void dump_stack(void)
{
        int i;

        printf("Stack contains: ");
        for (i = 0; i < STACK_SIZE; ++i)
                printf("%4d", stack[i]);

        printf("\tsp1 = %lu, sp2 = %lu\n",
                (unsigned long)stack_ptr1, (unsigned
long)stack_ptr2);
}

/* End of File */
```

C Express: 250+ Ready-To-Run Assembly-Language Routines for Turbo C, Microsoft C, and QuickC

C Express: 250+ Ready-To-Run Assembly-Language Routines for Turbo C, Microsoft C and QuickC
Robert Jourdain
Brady Books, 1989
ISBN 0-13-933185-9

Reviewed by Stephen Patten

C Express: 250+ Ready-To-Run Assembly-Language Routines for Turbo C, Microsoft C and QuickC, a book and accompanying diskettes, was written (both the assembly source code and text) by Robert Jourdain and published by Brady Books, a division of Simon & Schuster, Inc. The copy I reviewed was dated 1989.

Companion Diskettes

Two 5.25" diskettes accompany the 412-page book. One diskette contains the eight library files which house the 250 plus routines in object code form along with eight parallel header files for function prototyping. The other diskette contains the assembly language source of the routines in compressed form. (You can exchange the 5.25" disks for 3.5").

You install the libraries and header files on a hard drive by a simple copy, and then call routines as you would any C function inside source code. A global variable is also set which is used by the compiler to configure the correct memory model.

Book Contents

Chapter 1 provides a short introduction to using the library. Chapters 2-9 document the library files included on the companion disk. Chapter 10 includes a discussion on writing your own assembly subroutines to link with C programs. This is followed by indices of general topics, the routines, and their object files.

The routines or functions are documented pretty much as you would expect, each described in terms of its purpose, parameters, globals required, kinds of errors checked, and relevant peculiarities. Examples of calls are also amply provided.

The book also presents background information on groups of routines. For example, functions that implement expanded memory are preceded by a description of the LIM specification, not in great detail, but enough for the programmer to understand what the routines do and why.

Stephen Patten is a senior analyst with the Lincoln Savings Bank in New York and teaches C at New York University. He can be reached at (516) 932-3484.

Interestingly, video routines which can be implemented in either ROM BIOS or memory-mapped form are presented both ways, with the ROM BIOS routines suffixed with an underscore and lower case *b*. Depending upon the application, the programmer can choose between speed and portability.

Overview of Functions

The equipment configuration routines access the usual details of installed hardware and are only notable in that they eliminate the bit manipulations required when using the compiler library to extract the same information.

The memory management routines test for extended and expanded memory, and allow programming of expanded memory applications. Pages can be allocated, switched and deallocated, and data exchanged between expanded memory and RAM quickly and easily. This can be a real plus to a C programmer restricted by the one megabyte limitation of the MS-DOS environment.

C Express routines, like *hex_to_char*, *hex_to_int*, *binary_to_char*, and *binary_to_int*, interpret such strings as numbers, producing values which can then be entered as numeric data in C programs. Combined with existing *stdlib.h* utility functions, like *itoa*, they can also be used to convert binary to hex, or vice versa, a nice library facility.

There are string manipulation routines to add or delete characters to or from strings, change string characters, and perform various substring operations. There are also keyboard routines that provide for fast operating system calls to check, read, clear, wait on and change characters, and scan codes in the keyboard buffer.

filter_in, for example, captures the next keystroke in the buffer then looks for the same character in a predefined table. If found, the keystroke is accepted; otherwise, it's rejected. A related routine, *filter_out*, does the opposite, rejecting a character found in the table.

key_pause allows a key to act like an ON-OFF switch. Hold it down and, let's say, a help screen appears. Release it, and the help screen disappears. This can replace long and complicated *if* and *switch* statements with cleaner, faster code.

Mouse routines, based on Microsoft's driver, position the mouse cursor, turn it on and off, capture a button press, report mouse positioning and control mouse motion to cursor movement. With *pixel_ratio*, you can set the exact movement ratio between the mouse and its cursor motion. With

`define_graphics_cursor`, you can set the precise pixel or "hot spot" inside the pattern of pixels that fall under the cursor display.

Screen control routines set the video adapter, position the cursor, set mono or color attributes, and control the writing of strings to the screen. For example, `clear_blink` changes the 7 bit in the attribute byte so that it controls the intensity of background colors rather than blinking, effectively doubling background colors in CGA text mode. `set_blink` restores the bit to its original interpretation.

`display` and `display_b`, unlike `printf`, are passed arguments to both position a string before displaying, and display with an attribute.

The graphics in C Express are limited to the CGA text mode. Like the screen control routines, there are both memory-mapped and BIOS versions. They draw the usual assortment of graphics figures as well as scroll sections of the screen either vertically or horizontally. Written in assembly, they run very fast.

The file routines are built around a data structure called `tree_array` which holds directory names and pointers to subdirectories. The routines work with the structure displaying a directory tree or searching through the tree for a file. It's a little tricky going at first, but the book makes it all pretty clear.

Finally, a variety of printer output services are included. They simplify sending output to the printer inside a C program, do a certain amount of printer formatting, interpret embedded control codes in strings and implement the BIOS print screen function. `wrap_line` formats a string for printing in word wrapped form, while `justify_line` formats a string in right-justified, word wrapped form. The `prtsc` routine performs a screen dump, sending both text and graphics images to the printer.

In conclusion, the C Express package provides a collection of object code files and routines for calling within a C program. The routines are not unique in terms of their functionality but are a fairly standard collection that can be found in other off the shelf libraries. They are easy to set up and use, and they run fast. I was unable to find any bugs. The book provides good, well-organized documentation, and guidance for using the library files.

Particularly for new C programmers, the package would very likely make an excellent starter kit and base from which to build their own language libraries. □

It's Back!

It's official. Paul Vixie has indeed taken over the moderation of *comp.sources.unix*. He also has the assistance of two co-moderators, Mike Stump and Nick Lai, to help him prevent the kind of backlog that occurred recently. Submissions to the news group, and comments to the moderators should be sent to *unix-sources-moderator@pa.dec.com* or *decwrl!unix-sources-moderator*. The queue has been cleared, and the backlog posted. However, it includes over six megabytes of sources in 37 separate postings. Here are just the highlights.

The first posting of the "new era" was *chop* from George Sicherman <*gls@hrmso.att.com*>. *chop* extracts selected fields or columns of lines from the specified files or standard input, and writes them to standard output. It is similar to *cut*, a tool from the UNIX Documenters Workbench. *chop* allows for variable field separators and for output of the fields in the order specified on the command line. *chop* is Volume 25, Issue 1.

Victor Abell <*abe@mace.cc.purdue.edu*> has obsoleted two of his older postings, *ofiles* and *fstat*, with a new version of *lsof*. LiSt Open Files displays the names of files opened by processes on selected UNIX systems. This new version in Volume 25, Issue 2, supports any file system based on the SunOS *vnode* model. This includes SunOS, AIX, HP-UX, NeXTStep, Dynix and some others. It understands most *vnode* extensions, NFS connections, FIFOs, multiplexed files, and UNIX and INET sockets.

Abell having obsoleted *ofiles*, Robert Ehrlich <*ehrlich@margaux.inria.fr*> then went and reworked the original *ofiles* and released *ofiles2* for Volume 25, Issue 72. This new version is more portable, but is still designed for BSD derived UNIX systems. As with *lsof*, *ofiles2* displays the names of files that are being used by processes.

A major update to the "Threaded RN" Newsreader was issued by Wayne Davison <*davison@boreland.com*> for Volume 25, Issues 4-16. TRN is based on RN 4.4, and the posting includes both RN and the changes to make TRN from RN. TRN uses the References header to build a discussion thread out of the news group. The articles are then presented in discussion thread order.

One of the long promised postings, held in the queue for ages, is *ease* v3.5. *ease* is a decompiler that converts *sendmail.cf* into a language that is much easier to understand, and then allows editing of that intermediate language. It also includes a compiler to convert the *ease* language back into a *sendmail.cf* file. Considering how unreadable *sendmail.cf* files are, anything is an improvement. However, *ease* is a vast improvement. Bruce Barnett <*barnett@crdgw1.ge.com*> contributed *ease* for Volume 25 Issues 17-22.

Wen-King Su <*wen-king@vlsi.cs.caltech.edu*> contributed *fsp*, one of the more interesting postings. It consists of a set of programs that implement a public-access archive server similar to *anonymous-ftp*. While the actual code is probably not of much use to most CUJ subscribers, the concept and the implementation of that concept are a very well thought out lesson in client/server computing. It shows the tradeoffs between multiple servers, and a single stateless server. FSP was posted as Volume 25, Issues 24-26.

Sydney S. Weinstein, CDP, CCP is a consultant, columnist, author, and president of Datacomp Systems, Inc., a consulting and contract programming firm specializing in databases, data presentation and windowing, transaction processing, networking, testing and test suites, and device management for UNIX and MS-DOS. He can be contacted care of Datacomp Systems, Inc., 3837 Byron Road, Huntingdon Valley, PA 19006-2320 or via electronic mail on the Internet/Usenet mailbox syd@DSI.COM (dsinc!syd for those who cannot do Internet addressing).

Several of the newer UNIX shells provide interactive command line editing. The *ile* program from Robert C. Pendleton <*bobp@hal.com*> provides this same capability for any program. It runs the program as a subprocess so it does not need to modify the program, nor does it even need the source of the program it is servicing. It cannot provide a command history from program to program however. *ile* is Volume 25, Issue 29.

One of the nicer capabilities of the Bitnet sites is the *listserv* program provided to automate maintenance of mailing lists. Anastasios Kotsikonas <*tasos@cs.bu.edu*> has provided similar capabilities in his *listserv* v5.31 package posted in Volume 25, Issues 35-40. *listserv* provides support for *list*, *list-owner*, and *list-request* aliases for the mailing list and provides the back end programs for archives, moderated lists, peer lists, automated subscription, changes of address, and canceling of subscriptions.

I get spoiled by running on a bit-mapped screen with a good windowing system. Just writing this column I have several windows open, one for the column, one for the list of files to write about, and one containing the file I am reviewing. For those stuck on standard ASCII terminals Juergen Weigert <*jnweiger@immd4.informatik.uni-erlangen.de*> has released version 3 of his multi-session package, *screen3*, for Volume 25, Issues 41-48. It allows several different virtual screens on a single terminal using a short "hot key" sequence to switch between the sessions.

One of the more popular UNIX shells for interactive use has been *csh*, the c-shell. Over the years, many interactive extensions to *csh* were published as *tcsh*. These extensions used to require the source of *csh* as the base for the patches. Now with version 6, the full source of *tcsh* can be released. *tcsh* is a "kitchen sink" shell. It supports command line editing, command and file name completion, lists, manual lookup, job control, and has been ported to many different UNIX systems. For those that prefer *csh* to *ksh*, then *tcsh* will give you all the benefits of *ksh* with the *csh* syntax. The newest version, 6.01, was contributed by Christos Zoulas <*christos@ee.cornell.edu*> for Volume 25, Issues 54-71.

Another major update is the latest version of the Revision Control System. RCS v5.6, which is a very flexible source code librarian not only supports source files, but also can track changes to binary files. RCS v5.6 was contributed by Adam Hammer <*hammer@cs.purdue.edu*> for Volume 25, Issues 77-87. New in v5.6 are changes to fix security problems, efficiency changes for retrieving older versions, and the following of symbolic links instead of breaking them, and more reliable lock files under NFS.

Emmet Gray <*fthood!egray@uxc.cso.uiuc.edu*> has updated his *mtools* package to version 2.0.5. *mtools* allows UNIX systems read, write, and manipulate files on an MS-DOS filesystem (typically a diskette). It emulates the commands *ATTRIB*, *CD*, *COPY*, *DEL/ERASE*, *DIR*, *FORMAT*, *LABEL*, *MD/MKDIR*, *RD/RMDIR*, *COPY*, *REN/RENAME*, *TYPE*, and *COPY*. The *FORMAT* command only adds the MS-DOS file system to the diskette. It depends on the UNIX low-level format routines to actually low level format the diskette. *mtools2* was posted in Volume 25, Issues 97-99. Volume 25, Issue 103 is a set of patches to *mtools* 2.0.5 from Henry van Cleef <*vancleef@netcom.netcom.com*> to support XENIX 286 systems. There were portability problems in the original release in regards to assuming that *ints* are at least 32 bits long. In XENIX 286, an *int* is 16 bits.

Recent patches appearing in *comp.sources.unix* include:

psroff had patches 5, 6, and 7 posted as Volume 25, Issues 32, 33, and 104. *psroff* allows both older C/A/T *troffs* and the newer *di-troffs* to work with Postscript printers and with HP Laserjet printers. Patch 5 is minor fixes mostly for compilation on 80286 style machines. Patch 6 is very important as several major features were broken and fixed by this patch. Patch 7 is fixes for *groff/di-troff* users using HP laserjets.

pathalias, the USENET map routing program, had patch 10 released by Peter Honeyman <*honey@citi.umich.edu*> as Volume 25, Issue 89. This is purely a bug fix patch and it is very small, but since so much depends on *pathalias*, I though it worth mentioning.

No Reviews

The status postings show several projects in the re-review stage, but nothing appeared in *comp.sources.reviewed* over the past two months. Now that *comp.sources.unix* is back, this is not unexpected.

misc Tries an Experiment

Kent Landfield, the moderator of *comp.sources.misc* tried an experiment this time. He took a very large posting, made a *tar* file of it, and then compressed the *tar* file. He then uuencoded the compressed *tar* file and posted that. Even posted that way it was 42 parts (plus part 00 with the text on what the

posting is and how to convert it back to a *tar* file). If posted as the normal *shar* format it would have been over 140 parts, probably the largest single posting ever tried. Reviews of this method were mixed, with a large amount of complaints on how it is harder to determine what is there, and to sort and deal with it, but I had no problems saving the files, running my concatenation script and feeding that to uudecode to convert the ASCII back to binary. It then uncompressed cleanly and produced a 7.5MB *tar* file.

And what was this grand experiment..., *pp* v6.0, a Mail Transfer Agent (MTA). *pp* is designed for high volume message switching, protocol conversion, and format conversion. *pp* supports X.400, RFC-822, and RFC-1148bis conversion between RFC-822 and X.400. PP is designed as a replacement for MMDF or sendmail. *pp* speaks X.400 (1984 and 1988), SMTP, JNT, UUCP, DECNET Mail-11, X.500, alias files, RFC-822 local delivery, and Fax Internetworking. No User Agents are provided, but a line oriented and an X-Window based management console program are provided. *pp* v6.0 was contributed by Steve Hardcastle-Kille <*S.kille@cs.ucl.ac.uk*> for Volume 27, Issues 24-66.

The shadow log-in suite for UNIX systems was rereleased by John F. Haugh II <*jfg@rpp386.cactus.org*> for Volume 26, Issues 54-64. New in release 3 is support for SVR4 style maintenance utilities and the grouping of the code into libraries to make maintenance easier. This suite provides shadow log-in/password management to many UNIX systems that do not yet have such support natively. One file was left out of the distribution and was posted as patch 1 in Volume 26, Issue 75.

Robert Davies <*robert@am.dsir.govt.nz*> contributed a new version of his C++ matrix package for Volume 26, Issues 87-91. *newmat04* supports matrix, upper and lower triangle, diagonal and symmetric matrices, row and column vectors and the element type float/double. Operators include *, +, –, inverse, transpose, submatrix, determinant, decomposition, triangularization, eigenvalues, sorting and fast Fourier transforms. It is intended for matrices in the size range 4x4 to 90x90.

Have an HP Laserjet with the Pacific Data Systems 25-in-One font cartridge? If so, Bill Walker's <*bkw@uecok.ecok.edu*> *wroff* is what you need. It is a text formatter in the spirit of *nroff*, but designed specifically for this combination of hardware. *wroff* runs on UNIX, XENIX, MS-DOS and CPM-68K. It does kerning and some other *troff* like items as well. *wroff* is Volume 26, Issues 97-101.

Ted Campbell <*tcomp@hercules.acpub.duke.edu*> contributed *sfs* the space flight simulator for IBM PC's with EGA or VGA, UNIX- PC's with MGR or UNIX with X11. A 21-part posting in Volume 27, Issues 1-21, *sfs* offers a graphics-based real-time animated simulation of orbital flight. You can simulate a complete range of orbital parameters and can also simulate multiple planets in a solar system. A particularly full map is given of the earth and can be displayed as viewed from the orbiting spacecraft, as a ground track map, or as a distance perspective in which the orbital track can be seen.

A compatible *regex* (regular expression) package without any licensing restrictions was contributed by Tatu Ylonen

<*yloOngs.ft*> for Volume 27, Issue 23. It is fully compatible with the GNU *regex* library and can handle arbitrary data including binary patterns. It also can compile and run on 16-bit machines such as MS-DOS.

Those interested in genealogical research may want to get Steve Murphy's <*murf@oakhill.sps.mot.com*> *gcom* from Volume 21, Issues 72-78. *gcom* reads in GEDCOM format files containing genealogical data and merges them, utilizing not only name and date match heuristics, but familial ties as well.

In my February column, I mentioned Archie, the service that keeps track of which sites archive which data. Brendan Kehoe <*brendan@cs.widener.edu*> has updated his *archie* client in Volume 27, Issues 79-84. Version 1.3 of *archie* is his Prospero client for the *archie* service. Note, using this client requires TCP/IP access to an Archie server, which means you must be on an Internet.

• Last on the new release front is the latest release of *dmake*. Version 3.8 replaces version 3.7 and hopefully addresses all the little obscure bugs and features that remained. Dennis Vadura <*dvadura@plg.waterloo.edu*> has provided this version of the *make* utility. This package is the extended make *ala* BSD 4.4 release including many more features than the traditional versions. It includes support for UNIX, XENIX, MS-DOS, OS/2, and Atari-ST TOS. *dmake* 3.8 is Volume 27, Issues 101-142.

On the patch front, patch 9 was issued for *parseargs* in Volume 26, Issues 65 and 66 by Brad Appleton <*brad@ssd.csd.harris.com*>. *parseargs* provides a set of functions to parse command-line arguments. It can do much more than the *getopt* variety of parser. Patch 9 is mostly bug fixes and was followed by patch 10 in Volume 26, Issue 116 for a bit more cleanup.

qbatch from Alan Saunders <*tharrlalan*> had its patch 2 posted for Volume 26, Issue 70 and patch 3 posted in Issue 85. Again, these were bug fix releases.

The KSH lookalike, *pdksh*, posted by Simon Gerraty <*sjg@zen.void.oz.au*> was updated to patch 1 in Volume 26, Issues 71 and 72. The build process was cleaned up and some portability issues were addresses.

The tin threaded newsreader had patches 6 and 7 posted by Iain Lea <*stevax!iain*> in Volume 26, Issues 76-82. New are support for Minix 386, more *-M* options for *From* lines, unread articles, and scrolling, plus some bug fixes. Patch 6 was in five parts and patch 7 in two parts.

PBMPLUS the multi-format image manipulation toolset was also updated to fix some bugs and to add several new programs. Jef Poskanzer <*jef@well.sf.ca.us*> provided patch 10 in Volume 26, Issues 106-110. New are *pgmcrater*, *ppmforge*, *ppmtoacad*, and *sldtoppm*.

Alternative Games

Two different versions of a "get the money game" called *sokoban* were contributed by two different authors. Kevin Solie <*kevins@ms.uky.edu*> contributed his *xsokoban* for Volume 13, Issues 1-2. This X-based game is a very incomplete implementation of an idea from a similar PC game.

The other version, *xsokoban2*, is from Joseph Traub <*jt1o+@andrew.cmu.edu*> and was released in Volume 13, Issues 13-15. It provides a slightly different user interface than Kevin's version, but is essentially the same game.

Volume 13, Issue 3 provides *dr_mario* from Scott Noecker <*noecker@seq.uncwil.edu*>. This is a one-player lookalike version of Dr. Mario, a popular game for Nintendo that has nothing to do with the Mario Brothers series. It uses the standard keyboard and curses for the display driver.

A complete revision of the WCST Nethack spoilers file updating it to version 7 was contributed for Volume 13, Issues 4-9 by Paul Waterman <*wheaton!water*>. A complete reformatting has been done to make the information easier to use, and of course, more changes and sections have been added.

A connect-five-in-a-row-game, *xmake5* was contributed by Chih-Hung Hsieh <*hsiehch@spunky.cs.nyu.edu*> for Volume 13, Issues 10-12. This game, written in C++ provides both a curses and an X-Window interface. The X version uses the athena widgets library.

For those with multi-user networks, a multi-player networked bridge game was contributed by Matthew Clegg <*mclegg@cs.ucsd.edu*>. *okbridge*, Volume 13, Issues 16-23, is a computer mediated bridge game that allows four players to participate in a game of rubber or duplicate bridge. The program handles dealing, scoring and communication of bids and plays. Issue 23 is a patch to fix a small problem in one of the distribution files.

Previews from *alt.sources*

Its been a quiet two months in *alt.sources*, only 4MB worth of notable postings (that or I am being more selective in what I consider notable). The most notable posting is a release of a *uucp* work-alike package from Ian Lance Taylor

<*ian@airs.com*>. *taylor-uucp* v1.01 was posted on November 24, 1991 in 18 parts. It is a complete replacement for HDB style UUCP. It supports V2 style configuration (L.sys, L-devices) and BNU (aka HDB) style configuration files (Systems, Devices). It is a complete system, except for the fancy maintenance shell scripts and *uusched* (the latter is in the works).

Curt Mayer <*hoptoad!curt*> posted his disassembler for Z80/Z280 CP/M .COM files on November 27, 1991. The output is capable of being assembled and can detect code sequences by tracing jump targets.

Along with the CP/M disassembler, D'Arcy J. M. Cain <*druid!darcy*> posted on December 19, 1991 in three parts his Z80 CP/M emulator for UNIX. Most of the Z80 instruction set is implemented, except for the I/O section. The emulator also uses the UNIX shell commands to simulate some of the CP/M commands and uses the UNIX file system for drives. All I/O must go via the BDOS or BIOS as IN and OUT opcodes are not fully implemented.

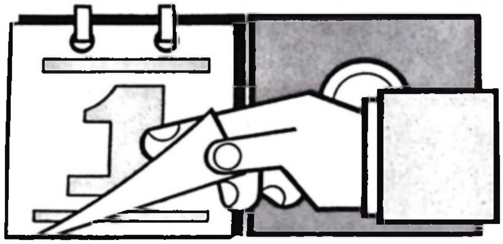
On the opposite front, Quinn Jenson <*jensenq@qcj.icon.com*> posted a DSP56001 assembler on November 29, 1991 in four parts. The syntax was intended to be compatible with Motorola's syntax, but without the docs he could only guess. It does allow for UNIX based DSP code development for those not lucky enough to have a NeXT.

pcal, the postscript calendar program has been updated to Version 4.3 by Joseph Brownlee <*jbr0@cbnews.cb.att.com*> on December 16, 1991 in seven parts. *pcal* allows for creation of personal calendars. New in 4.3 are generation of UNIX calendar files, move the previous and following month boxes around on the page, allow notes in any empty day box, change both note font and size via command line options, addition of the nearest keyword as in "workday nearest every 10th", plus about a page more worth of additions (something had to fill the 13000 lines of the posting).

Mayan Moudgill <*moudgill@cs.cornell.edu*> has posted his C++ socket library that adds UNIX and INET sockets to the *iostreams* functions. It was posted in late December, and then an updated posting was made on January 8, 1992 in two parts. Supported are convenient connection setup routines, use of the usual << *&* >> operators for i/o, support for out of band message transmission and reception, and an easy method of specifying per socket SIGIO, SIGURG and SIGPIPE handlers. Also include is an interface to the select system.

A gateway between the UNIX and the Citadel style BBS was posted on December 24, 1991 in four parts by Ken MacLeod <*unidel@bitsko.slc.ut.us*>. *uccico* implements the Citadel BBS networking for a UNIX system much like *uucico* does for UUCP. It handles USENET news and RFC822 mail conversion.

A large posting was *mixview*, posted by Robert Lau but written by Douglas Scott <*doug@woof.columbia.edu*> in 11 parts on January 12, 1992. It is an X-Window program that allows for editing sound files. It supports any BSD style system that has sound capabilities, such as Sun's and NeXT's. □



Calendar of Events

April

1-3 *Eleventh IEEE International Phoenix Conference on Computers and Communications*, Scottsdale, Arizona. Contact IEEE IPCCC-'92, P.O. Box 8950, Scottsdale, AZ 85252, (602) 234-4477.

6-10 *LATIN '92*, Internation Symposium of Latin American Theoretical Informatics, São Paulo, Brazil. Sponsored by ACM. Contact Univerisdade de São Paulo, Instituto de Matemática e Estatisticas, Caiza Postal 20570; 55-11-813-9499.

27-30 *XWorld, The X Info Xchange*, New York. Sponsored by the X Journal. Contact SIGS Publicatons, 588 Broadway, Suite 604, New York, NY 10012, (212) 274-0646.

27-May 1 *USE inc. Spring Conference*, San Francisco. Contact USE, Inc., P.O. Box 461, Bladensburg, MD 20710, (301) 699-9336.

30-May 2 *Independent Computer Consultants Association 15th Annual National Conference*, St. Louis, MO. Contact ICCA at (800) GET-ICCA or (314) 997-4633.

May

5-7 *European Direct Marketing Conference*, Brussels. Contact European Direct Marketing Association, 34, rue du gouvernement provisoire, B-1000 Brussels, 32-2-217-63-09.

June

1-4 *Object Expo*, New York. Sponsored by SIGS Publications. Call (212) 274-9135.

8-12 *C Plus C++...in Action*, London. Sponsored by Boston University. Contact Boston University, Corporate Education Center, 72 Tyng Road, Tyngsboro, MA 01879, (508) 649-9731.

10-12 *International Conference on Intelligent Tutoring Systems*, Montreal. Sponsored by ACM. Contact University of Montreal, 2900 boul. Edouard-Montpetit, Dept I.R.O. Montreal, Quebec H3T 1J4, (514) 343-7019.

July

12-17 *AAAI-92*, San Jose, CA. Tenth National Conference on Artificial Intelligence. Contact AAAI-92, 445 Burgess Drive, Menlo Park, CA 94025, (415) 328-3123.

14-17 *Object Expo Europe*, London. Sponsored by SIGS Publications. Call (212) 274-9135.

20-24 *Logic at Tver '92*, Tver, USSR. Sponsored by ACM. Contact University of Tver, 33 Zhelyahova Street, Tver 170013 USSR.

August

10-13 *C++ Technical Conference*, Portland, OR. Fifth annual C++ conference sponsored by USENIX. Contact USENIX Conference Office, 22672 Lambert St., Ste. 613, El Toro, CA 92630.

24-27 *18th International Conference on Very Large Data Bases*, Vancouver, BC. Sponsored by ACM. Contact University of Alberta, 615 General Services Bldg., Edmonton, Alberta T6G 2HI Canada, (403) 492-4589.

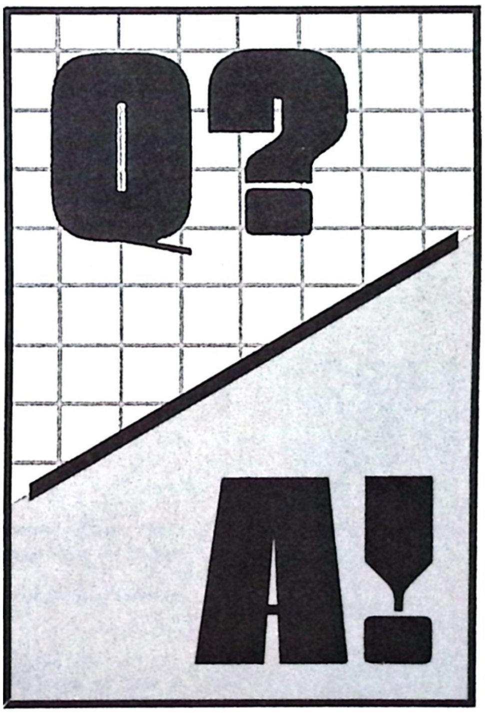
Calling Functions from Within a Function

Q I work with ANSI-C and I have a general problem. How is it possible to call one function with a variable number/types of arguments inside a second function with variable number/types of arguments, with the same arguments I called the second function?

For example, I have three functions with a variable number and variable type of arguments (*myfunction1*, *myfunction2*, *myfunction3* shown in Listing 1). I want to call the functions *myfunction1* and *myfunction2* from inside *myfunction3*, with the same arguments I called *myfunction3*. Is there an easy way to do that?

Thank you very much for your answer.

Willi Fleischer
Moerfelden, Germany



A The error involves the difference between a variable parameter list and a parameter list containing a value of type *va_list*. When you pass parameters to a function, the values of those parameters are pushed onto the stack. Usually the first or second parameter to a variable parameter function indicates directly (with a count) or indirectly (as with format specifiers) how many values have been passed.

The *printf* and *fprintf* functions expect to see the values on the stack. Each value has an address (its position on the stack). The *vfprintf* function expects its third parameter will be the address of the first value of a set of parameters on the stack. It then uses this address to retrieve those parameters. To use each parameter, it needs to know the type of the value. That information it gets from the format list specifiers, just like *printf* and *fprintf*. The type of the parameter is also used to find the next parameter.

Perhaps it might be instructive to look at typical definitions for these macros. Those in Listing 2 are from Microsoft C.

Variables of type *va_list* are actually addresses. *va_start* puts into the first argument the address of the first variable parameter. *va_arg* increments the first argument (*ap*) by the *sizeof* the type which is passed as the second argument (*t*), as well as yielding a value of that type.

How does *vfprintf* get to a value on the stack? Since it knows the type of argument (from the format list), it uses the *va_arg* macro with the appropriate type.

Your *my_function1* and *my_function2* require a list of values to be passed to them. When you used the *va_start* macro in *my_function3*, you retrieved the address of the first variable parameter that was passed to *my_function3*. You then passed that address to *my_function1* and *my_function2* and they produced garbage. You need to rewrite *my_function1* and *my_function2* such that they expect an address (i.e. type *va_list*). Listing 3 shows how they should look.

Q I've been programming in C for about two years now and have not seen anything in the literature about my question. The company I work for does quick and dirty production programming. We are in the process of moving from PL/I to C. The programming involves sequential processing of mostly fixed fielded files. In PL/I we read the file into an input buffer and use the *string* function to load it into a structure. Listing 4 is an instance.

Kenneth Pugh, a principal in Pugh-Killeen Associates, teaches C language courses for corporations. He is the author of C Language for Programmers *and* All On C, *and was a member on the ANSI C committee. He also does custom C programming for communications, graphics, image databases, and hypertext. His address is 4201 University Dr., Suite 102, Durham, NC 27707. You may fax questions for Ken to (919) 489-5239. Ken also receives email at kpugh@dukemvs.ac.duke.edu (Internet).*

What this in effect does is load the structure *recin* from the input buffer *inn*. There is no corresponding C function for loading such a structure and I've been trying to develop something that will do this. What I've come up with is Listing 5.

While the coding for *load_struct* works (at least in VAX C) I am not convinced this is the best way or even an effective way for doing what I desire. I realize the error checking is non-existent, but is it correct to assume that a structure of character arrays will have contiguous addresses? Is there another method used by more experienced programmers? I need something that I can put into a library that is very generic. How would you tackle this problem?

Tom Crosman
Brooklyn Park, MN

A On some machines individual fields in a structure do have packing bytes between them to align them to word boundaries, giving them contiguous addresses. Records which consist of character only (such as your example) tend not to have packing bytes.

Your method in general is fine. Since you asked for my solution, I've given it in Listing 6. Let me explain the modifications that I've made. The first is that one should usually never declare a variable in the same statement that declares the structure template. Anyone who wishes to use that template gets stuck with that extra variable. Second, I used an array of sizes for each of the fields. These could actually be picked up using the *sizeof* operator. As another alternative, one could *#define* the size of each field and use those in the initialization list.

I prefer using an array for the sizes instead of passing them in the parameter list. The declaration can be close to the structure template. Any changes in order or size of the fields can be simply coordinated. Using an array to pass the sizes also

Listing 1

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <stdarg.h>

void  myfunction1(char *format, ...)
    {
    va_list arg_ptr;
    va_start(arg_ptr,format);
    vfprintf(stdout,format,arg_ptr);
    va_end(arg_ptr);
    }

void  myfunction2(char *format, ...)
    {
    FILE *fp;
    va_list arg_ptr;
    fp = fopen("TEST.DAT","a+");
    va_start(arg_ptr,format);
    vfprintf(fp,format,arg_ptr);
    va_end(arg_ptr);
    fclose(fp);
    }

void  myfunction3(char *format, ...)
    {
    va_list arg_ptr1;
    va_list arg_ptr2;
    va_start(arg_ptr1,format);

    myfunction1(format,arg_ptr1);

    /* here I want to use the  arguments of myfunction3(),
   va_end(arg_ptr1); but this code does not work */

   va_start(arg_ptr2,format);
   myfunction2(format,arg_ptr2);

   /* here I want to use the  arguments of myfunction3(),
   va_end(arg_ptr2); but this code does not work */
   }

int main()
   {
   char  msg[]="message";
   myfunction1("\n%s=%d=%f", msg, 2, 3.0);  /* this works
fine */
   myfunction2("\n%s=%d=%f", msg, 2, 3.0);  /* this works
fine */

   /*  the following call of myfunction3() does not work.
   I want to have the same result, as
   if I call  myfunction1() and myfunction2() isolated  */

   myfunction3("\n%s=%d=%f", msg, 2, 3.0);
   }

/* End of File */
```

Listing 2

```
typedef void *va_list;
#define va_start(ap,v) ap = (va_list)&v + sizeof(v)
#define va_arg(ap,t) ((t*)(ap += sizeof(t)))[-1]
#define va_end(ap) ap = NULL
/* End of File */
```

Listing 3

```
void  myfunction1_a(char *format, va_list arg_ptr)
    {
    vfprintf(stdout,format,arg_ptr);
    }

void  myfunction2_a(char *format, va_list arg_ptr)
    {
    FILE *fp;

    fp = fopen("TEST.DAT","a+");
    vfprintf(fp,format,arg_ptr);
    fclose(fp);
    }

void  myfunction3(char *format, ...)
    {
    va_list arg_ptr1;
    va_list arg_ptr2;

    va_start(arg_ptr1,format);
    myfunction1_a(format,arg_ptr1);
    myfunction2_a(format,arg_ptr1);
    va_end(arg_ptr1);
    }
/* End of File */
```

Listing 4

```
DCL 1 RECIN,
    2 NAME    CHAR(30),
    2 ADDRESS CHAR(20),
    2 CITY    CHAR(15),
    2 STATE   CHAR(2),
    2 ZIP     CHAR(5);

    ...

    READ FILE (FILIN) INTO (INN);
    STRING(RECIN) = INN;
    ...
/* End of File */
```

Listing 5

```
#define MAX 1000

void load_struct(instruc, instr, ...);

/************** main function ********/

load_struct(instruc, instr, va_alist)
char *instruc;
char *instr;
   {
    int k=0;
    int strnglen;
    va_list ap;
    va_start(ap);

    while((strnglen = va_arg(ap,int)) != NULL)
      {
      for(k = 0; k < strnglen-1; k++)
         *instruc++ = *instr++;

      *instruc++ = 0;
      }
    va_end(ap);
    }
/* End of File */
```

Call for Papers

The C Users Journal is seeking articles on the topics below. If you have an idea for a related story, or experience that would qualify you especially to write on one of these topics, contact *The C Users Journal* editorial staff for *Author Guidelines* at:

The C Users Journal
Attn: Managing Editor
1601 W. 23rd St., Suite 200
Lawrence, KS 66046-2743
(913) 841-1631

We prefer practical treatments of real programming problems, or tutorials that make theory or complex issues more accessible to practicing programmers. Our emphasis is on C and C++ or on tools used in conjunction with these languages.

We pay at rates that are competitive with other national technical journals. We believe that our editorial assistance is better than most. You don't have to be a professional writer to meet our acceptance criteria, but you must have something to say. Again, contact the editorial staff for *Author Guidelines*.

Graphics

Proposals due 4/13/92; manuscripts due 5/11/92

Suggested topics: Writing C or C++ code that ports easily among the major GUI environments; useful graphic classes and libraries; algorithms for drawing, shading, windowing, scaling, etc. that are efficient and/or elegant; techniques for visualizing data; manipulating gray-scale and color images.

Overworked topics (require fresh perspective): graphic object classes, graphic file reformatters.

Communications and Networks

Proposals due 5/4/92; manuscripts due 6/15/92

Suggested topics: Interfacing C or C++ code to any of the popular commercial network packages; strategies for implementing simple but robust network protocols; programming popular ports or multiplexors in C or C++; tutorials on common interface protocols, interchange formats; user reports on major communications, interchange packages; simple handshake algorithms; real-life applications of queuing theory.

Overworked topics (require fresh perspective): UART drivers, kermit protocol.

simplifies the function, since it is no longer concerned with a variable parameter list.

To show an additional use for the array, I included a print function. It requires most of the same parameters as the load function.

To make the function more generic, I added a *nul_terminated* flag. Some programmers do not like using the extra character space to hold a terminator in each field. The *nul*, if present, can signify the end of a value less than the field length. If not present, the field length is the size of the field. Though this requires a slight bit more coding, it does save significant disk space if you store thousands of copies of a structure.

If you were concerned with the packing of the fields in either the input record or the output structure, then you could add an array of field addresses to the calls. If that is necessary, I might suggest not using a generic function and simply hard coding any record conversions required. At some point the work of providing and using a generic interface exceeds the benefit.

Reader Feedback

Coding style

I notice in your *find_maximum* function that you declare temporary storage for the return value. Is there some reason this is preferred over just returning the value directly? This allows the simplifications in Listing 7.

Also in your listings, for the function *put_line*, it seems to me that *putchar* would be preferred to *printf("%c",...)*. It would not have to process the format string and convert the character before putting it on *stdout*.

I enjoy your column and comments.

Edward C. Sarlls, III
Houston, TX

Listing 6

```
struct rec {
   char name[31];
   char address[21];
   char city[16];
   char state[3];
   char zip[6];
   };
static int field_sizes[] = {31, 21, 16, 3, 6};
/*
Alternatively, this could be written as:

static int field_sizes[] = {    sizeof(rec.name),
   sizeof(rec.address),
   ...
   };
*/
#define field_count (sizeof(field_sizes)/sizeof(int));

   main()
      {
      char record_in[MAX];
      struct rec record;
      ...
      while(fgets(record_inn, MAX, filin) != NULL)
           {
           load_struct(&record, record_in, field_sizes,
field_count, TRUE);
           print_struct(&record, field_sizes, field_count, TRUE);
           ...
           }
      }

print_struct(record, field_sizes, field_count, nul_terminated)
char *record;         /* Record to print */
int field_sizes[];    /* Size of fields */
int field_count;      /* Number of fields */
int nul_terminated;   /* If fields are nul terminated */
   {
   int field;
   int length;
   char *pc;
```

Listing 6 — Cont'd

```
   printf("\n");
   pc = record;
   for (field = 0; field < field_count; field++)
      {
      if (nul_terminated)
         printf("%s:",pc);
      else
         {
         length = field_sizes[field];
         printf("%.*s:", length, pc);
         }
      pc += field_sizes[field];
      }
   }

load_struct(record, record_in, field_sizes, field_count,
nul_terminated)
char *record;
char *record_in;
int field_sizes[];
int field_count;
int nul_terminated; /* If fields out should be nul terminated */
   {
   int field;
   int length;
   char *pc;
   char *pc_in;

   pc = record;
   pc_in = record_in;
   for (field = 0; field < field_count; field++)
      {
      length = field_sizes[field];
      strncpy(pc, pc_in, length);
      if (nul_terminated)
         {
         pc[length-1] = '\0';
         pc_in += length - 1;
         }
      else
         pc_in += length;
      pc += field_sizes[field];
      }
   }

/* End of File */
```

Listing 7

```
   int find_maximum(one, two, three)
   int one, two, three;
      {
      if (one > two)
           if (one > three)
                return one;
           else
                return three;
      if (two > three)
           return two;
      else
           return three;
      }

   ... or ...

   int find_maximum(one, two, three)
   int one, two, three;       /*  This could be a macro if you're
                                  careful about side effects. */
   return ((one>two)?((one>three)?one:three):((two>three)?two:three));

/* End of File */
```

Listing 8

```
int find_maximum(one, two, three)
int one, two, three;
    {
    int ret;
    if (one > two)
        if (one > three)
            {
            ret =  one;
            goto end;
            }
        else
            {
            ret = three;
            goto end;
            }
            ...
end:
    printf("find_maximum returning %d", ret);
    return ret;
    }
/* End of File */
```

Listing 9

```
if (one > three)
    {
    printf("Find maximum returning %d", one);
    return one;
    }

/* End of File */
```

Listing 10

```
void get_low_high(int a, int b, int c, int *low, int *high)
    {
    *low = (a < b) ? a : b;
    if (c < *low) *low = c;
    *high = (a > b) ? a : b;
    if (c > *high) *high = c;
    }

/* End of File */
```

I tend to use an automatic variable for the return value from a function. That makes it easier to put a `printf` statement in the code to print the return value of the function. Or if you are a debugger person, it makes it easy to watch the value.

The disadvantage is a slight decrease in speed. If I made it a register variable (or if the compiler does so automatically), even that should not be a problem.

If you decide to change the return value of the function that uses an expression, and then with the local parameter, you only have to change the expression in one place. If you have debugging output, you need to change it in two.

I must admit that I have had this style for a long long time. In one of my C classes that I taught back in the early '80s, I had a student who insisted that parentheses were required around the expression that follows the *return* statement. It turns out that all the examples of *return* statements he had seen had complex expressions around them (as the one in your second example). Psychologically the parenthesis were needed to surround the expression and "make it one." That points up the other coding style that I use quite often (see Listing 8). I state in *All on C* that having a single *return* statement with *goto*s is preferable to having multiple *return* statements. If I want to trace the `return value`, with multiple *return* statements, I have to do something like the code shown in Listing 9.

If I were using a debugger, there would be several breakpoints to set (assuming the function was long enough that I simply didn't single step through it).

The difference in the complexity of code between multiple *returns* and multiple *gotos* (to the end of a function) does not seem to be a big issue, at least to me.

I may get hundreds of letters regarding this seemingly idiosyncratic style of programming (or maybe with an adjective using only the first two syllables). Before that occurs, I wish to make a few caveats regarding it. First, I try to program the logic not to require multiple *returns/gotos*. Hence the original listing has neither in it. Second, there should be a single label at the end of the function with a standard name (say *end*), that is the label for the *goto*. If that is the case, then `ret = xxx; goto end` takes on the same meaning as `return xxx;`. (KP)

Q I just read your question and answer article entitled "Using *typedef*" in the December 1991 issue of *The C Users Journal*. I am writing in reference to the question regarding the writing of a function returning the lowest and highest integer out of the three integers passed. Listing 10 is my attempt at answering this question.

Listing 11

```
#include <stdarg.h>
int maximum(int count, int first,...);
main()
    {
    int ret;
    ret = maximum(6, 2,3,4,5,9,8);
    printf("Maximum is %d\n", ret);
    }

int maximum(int count,int first,...)
    {
    va_list arguments;
    int i;
    int value;
    int maximum = first;
    va_start(arguments, first);
    for (i = 0; i < count - 1; i++)
        {
        value = va_arg(arguments, int);
        if (value > maximum)
            maximum = value;
        }
    va_end(arguments);
    return maximum;
    }
/* End of File */
```

With all due respect, I think this approach is more eloquent and less convoluted than the methods offered in Listing 2 (December 1991 issue) and Listing 3 on page 120 (December 1991 issue) and in Listing 4 on page 122

Listing 12

```
void find_min_max(one, two, three, minimum, maximum)
int one, two, three;
int *minimum;
int *maximum;
   {
   if (one > two)
       *minimum = two;
   else
       *minimum = one;
   if (three < *minimum)
       *minimum = three;

   if (one < two)
       *maximum = two;
   else
       *maximum = one;
   if (three > *maxmium)
       *maximum = three;

   return;
   }
/* End of File */
```

(December 1991 issue), whether or not you choose to use the conditional shorthand.

I am also puzzled by the fact that you were a member of the ANSI C committee and you didn't use prototypes in your code. Did you have a special reason for not prototyping your examples?

S.J. Stern
Bothell, WA

Advertiser Index

This index is provided as a service to our readers.
The publisher assumes no liability for errors or omissions.

***This advertiser prefers to be contacted directly.*

Listing 13

```
    typedef double SPEED;
    typedef double DISTANCE;
    typedef double TIME;

    SPEED low_speed, high_speed;
    DISTANCE short_distance, long_distance;
    TIME brief_time, long_time;

/* End of File */
```

Listing 14

```
    SPEED compute_speed(DISTANCE distance, TIME time);

/* End of File */
```

A Eloquence is in the eye of the beholder. Some people might consider the second part of Listing 7 (from Mr. Sarlls letter previously in this column) the most eloquent. I don't particularly prefer any of my Listings 2, 3, or 4. My favorite is Listing 5 (from the December 1991 issue, reproduced here as Listing 11). It computes the maximum for any number of input parameters. The logic in your sample matches the logic in that listing.

I guess I could have arranged the logic in my function as shown in Listing 12. It matches your logic and has fewer lines than my previous listing. Unless I am going to call a routine a few thousand times, I tend to stick with whatever I come up with first that works. Also, I usually avoid using the conditional expression operator, for the reasons explained in last month's column.

As far as prototypes, I usually do not include them unless they are required by ANSI C or they are essential to the answer. The information contained in prototypes is mostly redundant. The case of the variable parameter function (as in Listing 10) requires one. When I do need prototypes (e.g. for C++), I let the compiler or PC-Lint generate a file of them.

Thank you for your feedback. (KP)

typedefs and *lint*

In my column a few months ago, I answered a question regarding *typedefs*. At the recent C-Forum sponsored by the Wang Institute of Boston University, I bumped into Jim Gimpel, the author of PC-Lint. He told me that the latest version of PC-Lint has an option for strong typing. This means that it can report errors in the use of *typedef*ed variables which the compiler would just ignore. For example, given the code in Listing 13, the assignment of

```
short_distance = high_speed;
```

is accepted by the compiler without question since both variables are declared to be type *double*, once the *typedefs* are resolved into the underlying types. However PC-Lint can yield an error message, if strong typing is turned on.

As another example, a function declared as shown in Listing 14 will give a PC-Lint error if it is passed the code in Listing 15.

There are many options available for strong typing, which are all described in the manual. In addition, you can declare that particular arrays can only be indexed by variables of particular types. For example, arrays of type *HISTOGRAM* can only be indexed by variables of type *INDEX*. This would look something like Listing 16.

Listing 15

```
low_speed = compute_speed(long_time, short_distance);

/* End of File */
```

Listing 16

```
/* lint -index(c,INDEX,HISTOGRAM) */

typedef unsigned int INDEX;
typedef int HISTOGRAM
#define SIZE (INDEX) 10

HISTOGRAM my_array[SIZE];
INDEX good_index;
int not_good_index;

my_array[good_index] = (HISTOGRAM) 5;
my_array[not_good_index] = (HISTOGRAM) 7;

/* End of File */
```

The *not_good_index* reference to *my_array* would yield an output error.

For those who are considering changing to C++ purely for its type-checking abilities, I suggest you look at PC-Lint as an alternative. (KP) □

Gadgets

Update

CUG297 Small Prolog

Henri de Feraudy (FRANCE) has released v2.0 of his Small Prolog. The new version offers much better debugging facilities, and a 32-bit executable compiled by GCC-386 (CUG359).

CUG327 Panels for C

J. Brown (KS) has released v2.3 update of his shareware window package, Panels for C. This version includes these new features: OS/2 support, Turbo C support, utilizing the PATH environment variables to find panel definition files, allowing the inclusion of panel definitions in the C source program, Interactive Panel Design (IPD) utility.

CUG351 UltraWin

Kevin L. Huck (MO) has released v2.10 of his shareware, Ultrawin. This new version includes new features: unlimited overlapping windows, background printing, PC timer control, mouse and graphic support, and enhanced data entry capabilities. Also included are a hypertext help engine and and EGA/VGA font editor. Also released is InTUItion 1.10, a textual user-interface library that includes an interface contruction program that allows UltraWin users to interactively create dialog boxes, menus, pick lists, forms and more using a mouse. Source code can be automatically generated to perform processing on each item, saving hours of tedious hand coding and debugging.

New Releases

CUG361 Gadgets and Term

Jack E. Ekwall has contributed a function library Gadgets, a group of UNIX-like tools for DOS; and Term, a collection of computer buzz-words. Gadgets provides functions such as popup/dropdown window, drawing box, screen and cursor manipulation, keyboard input, color, date, printer and mouse control, and file manipulation. Some of the functions are lifted from CUG273 Turbo C Utilities. The library is linkable to Turbo C v2.0. These UNIX-like tools offer a solution to the DOS command line interface pipeline problem. Term includes 634 topics and 32 historical notes/observations about computer buzz-words. This text is in a text-indexed sequential form which can be read by a display program, VU. The distribution disk includes source code for the library and documentation.

Kenji Hino is a member of The C Users' Group technical staff. He holds a B.S.C.S. from McPherson College and an undergraduate degree in metallurgy from a Japanese university. He enjoys playing drums in a reggae band.

User Report

Zinc Interface Library

Comments by David Brumbaugh

Introduction

The Zinc Interface Library is a C++ user interface library from Zinc Software Inc. for PC compatible computers. It supports MS-DOS text mode, MS-DOS graphics mode and MS-Windows 3.x interfaces. I'm reporting on Zinc Version 2.0 for Borland C++. Zinc also supports the Zortech C++ complier.

C programmers moving to C++ and experienced C++ programmers will find the Zinc Interface Library helpful. I've been using it for over a year on projects I've been working on at home.

Features

Zinc's primary feature is a C++ class library. It consists of class definitions, object code and optionally, source code for those classes. The class library is designed so that your application only needs to have one set of source code for writing applications in text mode for MS-DOS, graphics mode for MS-DOS (CGA, EGA, VGA, Hercules compatible) and MS-Windows 3.x.

The user interface created in all three modes is SAA compliant. This means it has windows, menus, dialogue boxes, list boxes, optional mouse support, and all the other features users have come to expect in modern software.

All three modes, text, graphics and MS-Windows have the same basic look and feel. The *UI_DISPLAY* class encapsulates the three display types in its descendants. The *UI_DISPLAY* class defines all the things that a program can do to a user's display.

Text mode applications use the PC extended ASCII graphics set for windows. The programmer has several options when using text mode:

1. The application can automatically detect the current text mode and use it. This is the default.
2. The programmer can explicitly use 25x80, 25x40 or 43x80 (which gives 50x80 on VGA).
3. The programmer can give the user a choice on which mode to use.

Graphics mode MS-DOS display classes use complier specific libraries. The library I have uses BGI (Borland Graphic Interface). The documentation indicates that there is similar support for Zortech's graphic library. An application can switch from graphics to text mode and back without loosing the information in the user's windows. Besides the text mode features, the graphics display classes have support for graphic specific features like arcs, polygons, bitmaps, etc.

Editor's Note:

As this issue went to press, we were informed that Zinc Interface Library v3.0 is now available. According to a Zinc representative, version 3.0 addresses some of the problems noted in this User Report. New features include direct use of Windows bitmap functions; MDI support for Windows and DOS; new window objects such as toolbar, combo box, checkbox, radio button, and buttons with associated bitmaps; Zinc Designer adds a toolbar and access to all library features including user functions and validation routines. For more information contact Zinc Software Incorporated, 405 South 100 St. Suite 201, Pleasant Grove, UT 84062, (801) 785-8900, FAX (801) 785-8996.

David Brumbaugh is a project manager at Advanced Information Services, asystems integrator in Peoria, IL. He has been programming in C for over fiveyears and in C++ for over a year. He can be reached by mail at 2807 N.Renwood Ave., Peoria, IL 61604.

While one MS-DOS application can support both text and graphics, you must recompile your program to support MS-Windows. You also must add a couple of `#ifdef` statements to call `WinMain` instead of `main`, and to redefine how colors are used. Other than that, all my MS-DOS applications, including graphic applications, ran without modification on MS-Windows.

The strong points of the class library far out-weigh the weak points. Because the class library is set up in a very logical hierarchy, it is easy to learn.

The field validation capability is one of my favorite features. All user input can be validated by the program. Most common validations are included and the programmer can define his own easily.

The library is fairly robust. Most of the errors I found in my programs were my own. When they weren't there was usually a fix on their BBS. The library is complete and generally well designed.

The only weak points that I found were in the MS-Windows mode. The first is that since it always displays its bitmaps one pixel at a time, bitmap displays in MS-Windows are very slow.

When I tried to work around this by using Windows bitmaps, I discovered that there is no obvious way to use Windows resources with Zinc. Zinc has its own version of resources, and the BBS contains some programs to convert Windows bitmaps to Zinc bitmaps and Windows icons to Zinc icons.

Finally, Zinc doesn't have any direct support for MS-Windows Multiple Document Interface (MDI). MDI applications keep all the child windows of a main window confined within the boundary of the main window. Some programmer's will see this as a bigger problem than others.

The other major feature that comes with Zinc is the Zinc Designer. The Zinc Designer is a program that allows the programmer to draw windows, dialogues, menus, bitmaps etc. Using the Designer is faster than writing the equivalent source code. It allows the programmer to make better "Look and Feel" decisions.

I found several weak points with the Designer. Some of the features that Zinc supports, like radio buttons and check boxes, are not supported by the Designer. The menu items in the Designer are cumbersome to edit. My final complaint is that the Designer generates only binary objects, not source code. I would like to see both.

Documentation

The Zinc documentation consists of three books: The Programmer's Guide, The Programmer's Tutorial, and The Programmer's Reference. It also includes a Quick Reference Guide containing a list of the most common constructors, flags, event information, and a class hierarchy.

The Programmer's Guide provides a good overview of the concepts in the Zinc Interface Library. It is a short book that hits the most important points of the library. It also contains a user's guide to the Zinc Designer.

The Programmer's Tutorial is a clear and simple book to help the programmer get started. It is short enough to stay interesting. That is a major accomplishment when you consider that it not only contains lessons on using the Zinc Library, but a C++ and object-oriented design tutorial as well. The examples are excellent. They are clear and many are useful building blocks for your own applications.

The Programmer's Reference is a well organized book that covers most of the classes that come with Zinc. It could be more complete in its handling of specific classes. For example, the width parameter in the Line method of `UI_DOS_BGI_DISPLAY` is ignored. I spent about two hours trying to find the bug in my code before I checked with technical support. They agreed that it should have been documented.

Support

I have found Zinc technical support to be absolutely terrific. I usually use the Zinc BBS for support. It is well organized and well maintained. It contains corrections, news, user contributions and additional examples. The message base provides contact with the people at Zinc and with other users. I usually search the message base and find the answer I'm looking for without having to post a question.

When I can't wait for the BBS I call Zinc's voice line. I've never had a long wait when I've called. The support people are very friendly, helpful and knowledgeable. They usually understood my problem better than I did and had helpful suggestions in addition to the answers to my questions.

Competition

Borland's ObjectWindows and Turbo Vision are the other C++ user interface libraries I am familiar with. Borland chose to use separate class hierarchies for MS-DOS text and MS-Windows user interface. That means that if you want to write one application for both MS-DOS and Windows, you have to write two separate programs.

ObjectWindows has more direct support for MS-Windows than Zinc. It includes classes that can use Windows resources, for example. But, it lacks the field specific editing features, like dates, that Zinc has.

Turbo Vision is a text mode user interface class library. It has features Zinc doesn't, like the `THistory` class that allows the user to keep a list of data entry choices. It also lacks certain features, like the extensive field support, that Zinc has. Turbo Vision also lacks the MS-DOS graphics support that Zinc has.

Generally, if you're doing strictly MS-Windows programming, Borland's OWL has a slight edge because it is strictly for Windows. Its bitmaps are displayed faster, it can use resources from Borland's Resource Workshop (or other sources) and there is full MDI support.

If you want to write strictly text mode programs, Zinc has a slight edge over Turbo Vision because it has more support for formatted and validated data entry fields. If you want to write MS-DOS graphics mode programs, Zinc wins hands down because neither Turbo Vision or OWL supports that mode.

If you want to write applications that can be used across all three platforms, I recommend Zinc because you'll be doing less rewriting of code. The Zinc library fulfills the promise of code reuseablity much better than either ObjectWindows or Turbo Vision.

Conclusion

The Zinc Interface Library is a good value. There is no doubt that Zinc will save time for C++ programmers who write programs for the PC. It lets you program for three PC platforms in the time it usually takes to write code for one. The examples of C++ code will be of great help to the new C++ programmer. The library is an example of good Object Oriented Design.

I have only two suggestions for improvements. I would like to see a more complete reference manual. I also would like to see the Windows library have more support for the features of Windows 3.x.

I would recommend the Zinc Interface Library to any C++ programmer who wants to write applications for the PC. □

Notice

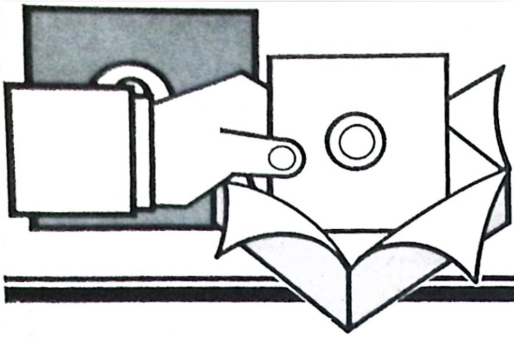
To Our Subscribers

Occasionally, *The C Users Journal* makes its mailing list available to vendors of products we think our readers will find interesting. Current subscribers receive free information in the mail from these vendors.

If you prefer that your name not be used in these mailings, please let us know. Just copy or clip this form and send it with your name and address to:

The C Users Journal

The C Users Journal
1601 W. 23rd. St., Ste. 200
P.O. Box 3127
Lawrence, KS 66046-0127



New Products

Industry-Related News
& Announcements

GUI Design Tool for OS/2 PM Adds 32-Bit C Code Generation

Gpf Systems, Inc. has released Gpf v1.3, an upgrade of its Gpf (GUI Programming Facility) tool for OS/2 Presentation Manager applications. The new version extends the platforms on which Gpf-generated applications can run by enabling developers to create native 32-bit applications. It also allows developers to define custom graphical objects for the user interfaces they design.

Gpf runs on OS/2 1.3 Standard Edition or later versions, and it supports Extended Edition for those applications using OS/2 DataBase Manager. The program requires a mouse and 1.5MB of available hard disk, and 6MB of RAM are recommended. Gpf is licensed on a workstation basis and is priced at $995 for the first copy. For more information *contact Gpf Systems, Inc., P.O. Box 414, 30 Falls Rd., Moodus, CT 06469-0414, (800) 831-0017, FAX (203) 873-3302.*

Quick and Cool PLD

Intel Corp has introduced a high-performance CMOS 22V10 compatible programmable logic device (PLD) running at 100-MHz count frequency with a 10 nanosecond (ns) propagation delay.

Intel's CMOS 85C22V10 PLD operates at 40 percent lower power than bipolar devices, thereby generating less heat. This translates into increased system reliability and decreased power supply requirements — important features for today's compact system designs.

Based on the largest-selling PLD architecture, the 85C22V10 is JEDEC compatible with standard 22V10's and is supported by current 22V10 software. Optional superset features included are programmable clock inversion and additional signal feedback configurations.

In quantities of 1,000 the PLCC and PDIP 10ns versions cost $9.50 and the 15ns versions cost $4.75. For further information *call a local Intel sales office, or the Literature Center at (800) 548-4725 (in the U.S. and Canada), or write for: Intel Literature Packet #IP-87, P.O. Box 7641, Mt. Prospect, IL 60056-7641.*

Softaid Introduces 15 Mhz Z180 Emulator

Softaid has released a version of its UEM In-Circuit Emulator for Zilog's 15 Mhz Z180 processor. The Z180 offers a Z80 processor core with an extensive suite of on-board high integration peripherals. Recently Zilog introduced a 15Mhz version of the Z180, giving embedded designers access to more computing horsepower while maintaining software compatibility with their older Z80 code.

The UEM includes 131,072 hardware breakpoints, a 4KB deep real time trace buffer, performance analysis, and a memory access monitor. It comes complete with a Source Level Debugger that supports essentially all C and PL/M compilers.

Softaid also offers a free Guide to Developing Z180 Applications, for those designers needing a little more information than provided in Zilog's sometimes cryptic data sheets.

The 15 Mhz Z180 UEM costs $6500.00 and is available from stock. For more information, *contact Softaid, Inc., 8300 Guilford Rd., Columbia, MD 21046. (800) 433-8812.*

Demo Maker/Player Programs for Software Systems Running Under X

Non Standard Logics, Inc. (NSL), has released a set of programs that allow developers working in X to create demos of their software quickly and inexpensively, with unlimited distribution of the demos at no cost beyond the initial purchase price.

Designated XDemoMaker and XDemoPlayer respectively, the two-program set employs an OSF/Motif graphical user interface, but demos produced are independent of the GUI and run under any X terminal graphical user interface used with the application. The program incorporates editing tools for fine-tuning the demos and for including text to amplify on the activities displayed on the screen. Complete or partial sequences can be repositioned within a demo by means of "Cut-Copy-Paste-Duplicate" commands.

XDemoPlayer is licensed to purchasers of XDemoMaker for use on specific platforms with unlimited copying rights, enabling developers to distribute their demos with the playing program in quantity. A standard 1.44MB diskette accommodates both the XDemoPlayer software and a demo of several minutes' length. Longer demos can be circulated on tapes and compact disks.

Single unit price for XDemoMaker is $6,000. XDemoPlayer is licensed for $4,000 for the first platform with discounts available for additional platform versions. Unlimited copying rights are included with the player license. The programs are now available for

IBM, DEC, Sun and Hewlett-Packard workstations. For more information, *contact Non Standard Logics, Inc., 4141 State Street, Suite B-11, Santa Barbara CA 93110, (805) 964-9599, FAX (805) 964-4367. In Europe, contact Non Standard Logics SA, 57-59 Rue Lhomond 75005 Paris, France, (33) 1 43 36 77 50, FAX (33) 1 43 36 59 78.*

Visual Program Design and Code Generation Tool for MS-Windows

ProtoView Development has released ProtGen 2.1, a visual program design and code generation tool for Windows application development. Similar to ProtoGen 2.0, sold by Borland International, version 2.1 features a menu designer and dialog linking interface. It also generates code for both ANSI C and Borland C++ Object Windows.

Using a C++ class object, ProtoGen 2.1 also allows C++ programmers to tap into the Proto View dynamic link libraries to access advanced data entry and validation capabilities. Masked input, date, currency, numerics and table selection controls and more, give a sophisticated look and feel with no coding effort.

ProtoGen's price is $199.00, but is now available at the introductory price of just $49.95. Upgrades for current users are also $49.95. It requires a PC with 80286 or higher processor running Windows in protected mode. Code generated by ProtoGen is royalty free. For further information, *contact ProtoView Development Corporation, 353 Georges Road, Dayton, New Jersey 08810, (908) 329-8588.*

New Release of Speededit

Bradford Business Systems recently released SpeedEdit vA.04, a programmer's text editor for MS-Windows, X-Windows, DOS, OS/2, MPE-XL and other operating systems and platforms. This version adds support for SunOS and Hewlett-Packard's MPE-XL. Under each platform, save MPE & MEP-XL, SpeedEdit comes in one of two forms, a character version to operate on standard terminals or non-windowed consoles and a Windowed version. The SpeedEdit system sells for $295 on all IBM-PC based platforms and $395 on single user UNIX workstations. Multi user UNIX systems are priced according to the system. For more information *contact Bradford Business Systems, Inc. at 23151 Verdugo Drive, Suite 114, Laguna Hills, CA 92653 (714) 859-4428, FAX (714) 859-4508.*

Multilingual Software Toolkit

Frontier Software Services has released the Translator's Apprentice, a programmer's toolkit which allows the programmer to write programs for multiple languages (French, Spanish, Russian, etc). Using the toolkit, programmers can create multi-lingual programs which start up in a user's choice of language and switch from one language to another at the press of a key.

The package simplifies the tasks of the analyst, programmer, and translator by providing a mechanism for the translation and update of textual material in an application without requiring changes to the program for each translation.

The toolkit costs $175 for the Text Manager and Access Functions, including source code and a 30 day money-back guarantee. Site licensing is available. For further information *contact Frontier Software Services at 31 Mystic Avenue, Winchester, MA 01890 (617) 729-2161.*

Debugging and Data Capture Tool

Paladin Software, Inc. is now shipping an upgraded version of DataScopeTM, the communications debugging and data capture tool with applications in the computer programming, manufacturing, industrial automation, and multimedia industries. Version 2.0 saves time and money, eliminates guesswork by allowing the user to apply powerful display and search tools to ordinarily invisible transmissions, and provides an alternative to expensive hardware line monitors.

DataScope is the only serial line monitor that includes context-sensitive Hypertext, Hypersetup, and user-alterable multitasking window displays. Version 2.0 offers a user-friendly, "windows-like" pull-down menu interface.

For further information *contact Paladin Software, Inc., 3945 Kenosha Avenue, San Diego, CA 92117, (619) 490-0368, FAX (619) 490-0177.*

Professional Input Validation for MS-Windows

MantaSoft Partners have released InControl, a new set of DLLs, with source code available, that will enable programmers to include professional input validation in their Microsoft Windows 3.0 applications. These DLLs are compatible with Microsoft Windows SDK, Borland C++ 2.0, Borland Turbo Pascal for Windows, Whitewater's Actor and Microsoft Visual BASIC.

InControl Toolbox defines thirteen new classes of controls for Microsoft Windows. The controls are broken into two logical groups: display and input. Display controls are used to display information to a user of a Windows application. Input controls are used to control the information that a user enters into a Windows application.

InControl Toolbox contains two display controls. These are the Time control and the Date control. It contains 11 input controls. The input controls are broken into three groups: Formatted, Numeric and Free Format. The Formatted group contains controls which allow the input of Zip Codes, phone numbers, Social Security numbers and dates. There is also a programmer defined Formatted control that allows the programmer to define, at a character-by-character level, the allowable input for a given control. This control uses a dBase-like command string to describe the allowable input, if any, at a given position in the input control.

The Numeric group contains controls that allow the entry of integers, floating point numbers and dollar amounts. Each of these controls allows the entry of negative numbers and has an option which allows the programmer to set a legal range of values.

The Free Format group consists of two general entry controls. They are called general entry because they accept any input and, upon losing focus, try to determine if that input corresponds to a legal value. The first of these controls is the Free Format Date control. This control will try to determine what date the user meant from the given input. For example: a value of "F392" would yield the result "February 3, 1992" or "02/03/1992" depending on a style flag. Other allowable inputs are "Next Week", "Last Friday", "Today" and "Yesterday".

The second general entry control is the Regular Expression control. This control uses a Grep-like regular expression to determine if the value intered by the user was a legal value.

For each of the input controls, a programmer can assign a function that is called when the control loses focus. This routine can be used to further validate the data entered or to immediately store the entered value into a variable in memory. This function notifies the control, through a

C// for C++

Subtlesoft announced C// for C++: the C extension injecting real-time parallelism into a single C program, now combining the power of C// and the flexibility of C++. An unlimited number of run-time created, separable processes smoothly cooperate on common resources, self-parallel functions, queues, lists, events, timeouts. All the C++ communications and classical synchronization mechanisms are available, along with a powerful set of new C// weapons, including the new C// class of semiautomatic variables, run-time control variables, double access to process arguments, stack monitoring, private stacks, processes with no stack, offsets, etc. Dynamic priorities and scheduling facilitate interprocess cooperation. Precise handling of external events through the C// driver, user-programmed ISRs, urgent process executions, and the complete resolving of the problem with DOS and BIOS non-reenterability make C// for C++ a useful real-time tool.

C// for C++ provides good memory management and performance. It provides a natural platform to pure object-oriented programming in addition to the C++ mechanisms. Real-time projects, communication and simulation software, internal multitasking environments, advanced control software are effectively implemented in C// for C++.

For further information *contact Subtlesoft International, 4344 Bristol Street, Pittsburgh, PA 15207; (412) 682-3934.*

return value, whether it considers the value to be valid or invalid.

Each control can notify the user of error conditions in two forms. A style can be set that causes an audible beep whenever an error is detected. Alternatively, a visual text message can be displayed immediately below the current input control which more fully describes the error. For example, if a user enters a '9' in a Formatted control in a location where the maximum value has been set to '5', the control would display a message informing the user of the range violation.

InControl Toolbox is available directly from MantaSoft Partners and retails for $179, or $249 with complete source code. Orders will also be taken by GUI Clearing House, which can be reached at 1-800-522-4624. For more information please *contact MantaSoft Partners, P.O. Box 203551, Austin, TX 78720, (512) 335-3497, CompuServe 70314, 1445.*

Programming Library With Added Memory Capability

Library Technologies announces the addition of EMS/XMS/virtual memory capabilities to their programming library C-Heap for the Microsoft C and Borland C/C++ compilers. The 51 new functions give the programmer low-, medium-, and high-level functions for utilizing EEMS 3.2, LIM 4.0, XMS 2.0, and virtual (disk space) memory, with no 64KB limit on memory block size. The most powerful, high-level functions provide an automatic swapping mechanism (with locking capability) to completely remove the burden of managing EMS/XMS/virtual memory from the programmer, making the use of EMS/XMS/vmem an extremely simple matter. At all function levels, the programmer specifies whether EMS or XMS is to be used preferentially, but the fucntions use whatever is available. If neither is available, or when EMS/XMS is used up, the medium- and high-level functions can use disk space automatically. The user of disk space can be selectively disabled, though, or memory can be allocated specifically from disk space if desired; thus, the programmer can be sure that memory is being utilized exactly as he wishes. All virtual memory I/O occurs through from zero to four disk caches of up to 16KB each in size, which the programmer may specify be allocated to dramatically increase the speed of virtual memory I/O. With the high-level functions, a "dirty bit" can be cleared to avoid writes to EMS/XMS/virtual memory when the data has not been changed, which can greatly speed execution. All functions are compatible with malloc(), and all memory models but the tiny model are supported.

These functions join the other 500+ assembly-language functions already in C-Heap, which deal with DOS memory management. The cost of C-Heap is $199, or $399 with source code. For further information *contact Library Technologies, P.O. Box 56031, Madison, WI 53705-9331, (800) 767-4214.*

Object Professional for C++

TurboPower Software announces Object Professional for C++, a powerful, time-proven library of text-mode user interface objects. Object Professional for C++ (OPC) is a straight port of the user interface objects from Object Professional for Turbo Pascal. OPC includes high-level objects such as text editors, dialog boxes, scrolling data entry screens, help systems, and more. The user interfaces are ready-to-use, and built-in calls allow programmers to customize the behavior. OPC does not use event-driven programming. Applications built using OPC will run nicely in 640KB 8088 based PC's.

OPC includes utilities to help build menu systems and data entry screens. The utilities support interactive design and testing and then automatically generate source code. OPC includes full source code, 1,300 pages of documentations, pop-up help, and plenty of example and demo programs. No payment of royalties is required. OPC requires Borland C++ 2.0 or 3.0 Object Professional for C++ costs $249. For further information *contact TurboPower Software, P.O. Box 49009, Colorado Springs, CO 80949-9009, (800) 333-4160.*

Shortcut for Rapid Application Prototyping

Cadre Technologies Inc. announces the availability of ShortCutTM, a rapid application prototyping tool based on Cadre's TeamworkR family of CASE products. Developed by Cadre's exclusive Austrian distributor, Computer & Software Engineering (C.S.E.), ShortCut incorporates a graphical user interface (GUI) editor, a database access library, a compiler, and a rule-based expert engine. Working together these components enable application developers to capture, analyze and validate end-user requirements before designing and implementing the actual production system. In addition they allow developers to generate a complete and functional application prototype and enable developers and potential end-users to interact with this prototype well in advance of the development of the code.

With ShortCut, developers can understand end-user requirements from three important application views: the user interface; functionality and control; and database access, which works with SQL databases such as Ingres, Oracle, and Sybase.

Priced at $7,995 for a base configuration that includes a GUI editor, a compiler, and an expert runtime system, ShortCut is available immediately on Sun, DEC, IBM, and HP workstations. For further information *contact Cadre Technologies, Inc., 222 Richmond Street, Providence, RI 02903, (401) 351-CASE, FAX (401) 351-7380.*

Database Independent Development Environment

Convergent Solutions, Inc. (CSI), has released its database independent development environment, CS/ADS Release 6.3, for use with ShareBase Database Servers attached to Sun 4 and Pyramid MisServer computers.

CS/ADS is a tool set for professional programmers. It provides a comprehensive high-productivity development kit for building complex business applications and informations systems. A significant benefit of CS/ADS is that it allows applications to be built that are database independent, yet still incorporate the full power of SQL for the specific RDBMS.

CS/ADS applications that are built for ShareBase can be easily moved to other RDBMS's by using CS/ADS for ORACLE or CS/ADS for Informix.

The centerpiece of CS/ADS is a fourth generation programming language (4GL) that combines high-level constructs and powerful abstract datatypes with the programming structure of traditional third generation languages. The development environment includes a dynamic data dictionary that extends the data definition beyond the database schema by including editing, verification, and formatting rules for data fields. Also, CS/ADS comes with a suite of interactive, developer friendly utilities such as an advanced WYSIWYG screen painter and 4GL code generator, all of which are intergrated to both the data dictionary and the RDBMS schema.

CS/ADS for ShareBase III will be initially available for Sun 4 and Pyramid architectures. The software will be ported to other CS/ADS supported platforms later this year.

For more information *contact Convergent Solutions, Inc., 100 Metro Park South, Lawrence Harbor, NJ 08878, (908) 290-0090, FAX (908) 290-1494.*

OS9 for Motorola Single Board Computer

Microware Systems Corporation now offers an optimized version of OS-9 for the Motorola MVME167 single board computer.

This latest version of OS-9 for Motorola microprocessor-based products allows designers to take full advantage of the 32-bit M68040 processor, while providing support for the on-board serial, SCSI and Ethernet hardware.

The OS-9/167 Development Pak includes a number of new OS-9 device drivers for the next generation I/O peripherals included on the MVME167 board family. These include new SCSI drivers for the NCR 53C710 controller which support Common Command Set flexible and hard disk drives and tape units. Additional drivers are included to support the on-board real-time clock and new CD-2401 serial I/O controller.

Full Ethernet support is provided by means of the OS/9 Internet Support Package (ISP) and new device drivers for the Intel 82597 Ethernet Controller. ISP allows an OS-9/167 Development Pak system to remotely login and transfer files between an MVME167 and UNIX or DOS nodes on an Ethernet network. Support for BSD socket-based interprocess communication is also provided.

The OS-9/167 Development Pak also includes a full set of resident development tools designed to jump-start application development. These tools include a full K & R C Compiler, Macro Assembler and Linker, User State Debugger, (mu)MACS full screen editor, Shell Command Interpreter, and numerous utility programs.

OS-9/MVME167 is available in two versions. The OS-9/167 Development Pak includes the OS-9/167 Real-Time Operating System modules and device drivers as well as the full suite of development tools. Cost for the Development Pak is $3,000. The OS-9/167 Run-Time Pak provides only the OS-9 Real-Time Operating System modules and is intended to provide target system functionality. Quantity one pricing for the Run-Time Pak is $1,500. Contact Microware for multiple-copy licensing information. Both packages are available now.

For more information *contact Microware Systems Corporation, 1900 NW 114th Street, Des Moines, IA 50325-7077, (515) 224-1929, FAX (515) 224-1352.*

Program Language Translation

Shannon Associates' METAMORPHOSIS is a generic utility program which facilitates the transformation of any syntactically reducible character-oriented file to other forms. Given the syntactical definition of the source and target languages, METAMORPHOSIS will translate source programs from one language to another; ie: FORTRAN to Ada, JOVIAL or Ada, any language to C, dialect conversions, etc.. METAMORPHOSIS also functions as a custom compiler, assembler, macro processor, graphics language processor and report generator. Further, METAMORPHOSIS facilitates reformatting of data base files and analysis of natural language, grammar, sequential and parallel procedures and computational signatures.

In addition to generic METAMORPHOSIS, preconfigured METAMORPHOSIS translators including FORTRAN IV to C PL/I (Subset G) to C, and CMS-2M to Ada are available for immediate delivery.

Generic METAMORPHOSIS, FORTRAN IV to C, PL/I (Subset G) to C and CMS-2M to Ada sell for $387, $87, $87, and $134 each, respectively. All execute on the IBM PC, PC/XT, PC/AT and compatibles with minimum 416K RAM, monochrome monitor, two 360K bytes 5.25" DSDD floppy drives, optional printer and MS-DOS / PC-DOS 2.0 or revisions.

Contact J.H. Shannon Associates, INC., P.O. Box 597, Chapel Hill, NC 27514 (919) 929-6863.

New Name for Raima Product

Raima Corporation announces that they are changing the nomenclature of db_VISTA, the high performance database management system, to Raima Data Manager. This move is in line with the company's new corporate image campaign of relating its product names to their specific technology.

The Raima Data Manager DBMS is best known for its high performance in applications development, and for its combined database technology of relational and network models.

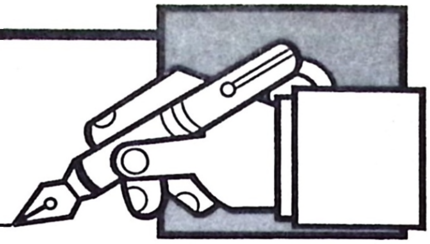
Raima began the new naming campaign last June when they introduced Raima Object Manager, a storage class library targeted in the C++ and object-orientated marketplace.

The name change to Raima Data Manager accompanies a new update release, version 3.21. The update does not present new features, but includes fixes to the current 3.2 version.

Raima Data Manager will continue to be marketed as both a stand-alone product, and as a full system accompanied by db_QUERY and db_REVISE. db_QUERY is an SQL-based query and reporting module; db_REVISE is a sophisticated database restructuring tool.

For further information *contact Raima Corporation, 3245 146th Place S.E., Bellevue, WA 98007, (800) 327-2462, FAX (206) 747-1991.*

We Have Mail



We ask that letters with code listings be submitted in an ASCII text file on an MS-DOS formatted disk. Providing us an electronic copy of the code will prevent typographical errors that might result from optical scanning or re-keyboarding.

Dear Sir,

I see that Randall Bart has raised the question of C++ vs. ++C. My own understanding is that the language is so named because every time you think you've finished learning it Bjarne adds a new feature.

Yours faithfully,

Scott Wheeler
BMT Research, Orlando House
1 Waldegrave Rd.,
Teddington, Middx
UK TW11 8L2

Wish I'd said that first. — pjp

Gentlemen;

It seemed good to take this opportunity to thank you for the work your staff has done this past year, and the appreciation many of us have (who don't write) for the quality of *The C Users Journal*. As a CPA, and accounting software developer, each new monthly issue frequently discusses and answers a current concern of mine. I can't tell you the number of times an article appeared at exactly the moment I needed the information. True, many articles are over my head and I marvel at how so much can be written about something I know so little about.

It also seemed good to nominate Mr. Leor Zolman as CUJ Writer of the Year. This in no way slights the efforts of the others, but my vote is based on the following criteria:

Appropriateness and general usefulness of subject: A.

Scope and technical considerations covered: A.

Ability to adapt subject material to other applications [portability]: A.

Tutorial content of material and usefulness in teaching C language concepts and capabilities: A

Humor and 'tongue in cheek' writing style (a spoonful of honey makes the medicine go down): A

Future subjects that would be of interest to me and perhaps others are suggested for your consideration, and include:

DOS TSRs: For example, how would one adapt Mr. Robert Bybee's article "A Portable VMS-Style Input Line Routine" to function as a TSR, similar to Peter Norton's NDE referenced in the article.

Keyboard I/O: A routine for polling the keyboard buffer. CUJ had an article on this subject some time ago, but it either did not go far enough or was beyond my ability to understand. The scope of the routine would be in two parts: (1) the ability to capture user keystrokes from an application and log them into an ASCII script file (perhaps subsequently edited by the user), and (2) the ability to call this routine and load the named script file into the keyboard buffer. In this way, users should be able to run repetitive applications or set up a series of applications (reports) to be chained together and run as a batch.

Batch File Compilers: Sometime ago I downloaded a batch file compiler from CompuServe, written by Douglas Boling (PC Magazine Ziff Communications). The language concepts I think that are required should make an interesting article. (I would love to know how this program works.) The input file consists of a standard W S batch file which is then compiled on the fly and a .COM file written to disk. How is this done?

Help Compilers: Beginning with standard documentation files (ASCII) as might be produced by a word processor, how would one build a compiler to translate ASCII files into an indexed binary format. The help utility program would access a file name passed on the command line, display an index for user selection, then handle paging and text display with keyboard control. Based on Leor's current article, this should be a piece of cake for him to take on as his next project. (You're welcome, Leor.)

Enclosed with this letter is my subscription renewal and disk order for processing.

Thanks again for your quality work in past years and may this year generate even more success for R&D Publications.

William C. Moench, CPA
North Gate Software
29204 Knickerbocker Rd.
Bay Village, Ohio 44140

It's always nice to hear when we do something right. Thanks for writing. — pjp

Leor responds:

Gee, if they'd given me report cards like that when I was a student at MIT, perhaps I wouldn't have dropped out! On the other hand, this kind of feedback probably means more to me than any real report card I've ever received. Thank you much.

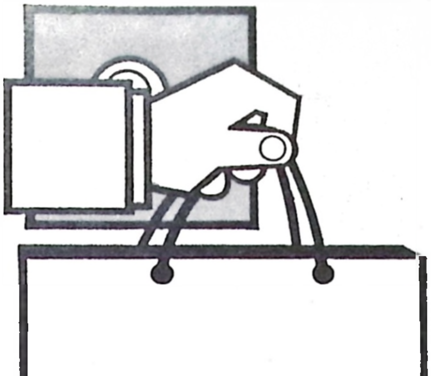
I find it difficult these days to sit down and write a program just as an intellectual exercise; usually, I have to have an actual need for the program before I'm motivated to write it (that's partially why I have resisted getting into into programming Windows; the other reason, a lack of elegant development tools, however, seems to finally be lifting...) Most of the material for Illustrated C *derives from applications I've written and put into service pretty much as they appear in the column and, now, the book as well.*

Incidentally, when bugs pop up, it's usually because the program hadn't gotten enough use yet. I'm too scatterbrained to debug by inspection; I have to debug by "using" — yes, otherwise known as trial-and-error. Plus, I'm a perfectionist. If nothing else, that makes for an entertaining development cycle. Well, in the immortal words of our distinguished publisher, "OK, I'm done."

Dear Mr. Phillips:

I very much enjoyed Part 5 of your series on Image Processing. It has stimulated some creative thought processes in me and for that I thank you. I did notice one problem with your examples which caused me to write this letter. The discussion below will clarify the situation.

The Kirsch, Prewitt, and Sobel transform sets are designed to detect edges in only one of the eight compass directions. As such, they can only be applied to the original image and can not be



Programmer's Market

C™ Programmer's Toolbox

MPW ($295); DOS Vol 1, 2 & 3 (Any volume $125; any 2 volumes $225; All $300); Sun UNIX ($495)

30 Day Money Back Guarantee

Dialects: Microsoft C6.x; MPW C; Think C; Turbo C; Unix C; ...

Unlimited number & sizes

CLint™ - Syntatically check one or more files & generate ANSI function prototypes; CFlow™ - Determine program organization, function and file interdependencies and runtime library contents; CPrint™ - Reformat/beautify C & C++ source code; CXref™ - Cross reference and check symbol usage, and identify unused files; CDecl - Translate and compose C declaration statements; CHilite™ - Highlight & print C source files (MPW only); pArc™ - File archiver with great compression (DOS only); pLn™ - UNIX symbolic links in DOS; PMon™ - Program profiler (DOS only); Cpp - flexible ANSI C preprocessor; plus many more...

SPECIAL

Buy 2 PC volumes and get 3rd free!

Call: (510) 770-0858
MasterCard Visa Accepted
MMC AD Systems
Box 360845
Milpitas, CA 95036

✧ Request 152 on Reader Service Card ✧

Wilsoft

Writebar Barcode Products

Wilsoft carries a complete line of bar code software and hardware for your every need. All major bar code types supported. Call the most experienced company in the bar code field today. DOS and Xenix/Unix support. Portable readers too! VISA/MC/AMEX

109 NW 22 Street
Ft. Lauderdale, FL 33311-3834
(305) 779-2720
(305) 763-3096 Fax
(800) 765-4114 Sales

✧ Request 115 on Reader Service Card ✧

PostScript From C!

c_pslib

- ✓Complete Set of Graphics Functions
- ✓Text Functions for fonts, lines, paragraphs
- ✓Downloadable Font support
- ✓Encapsulated PostScript File support

c_psFontInfo

- ✓Stringwidth Calculations
- ✓Font Reencoding
- ✓Kerning, Ligatures, & More!

Site License $595 each
Single User License $195 each
ANSI and K&R Source Code Included

Barton Creek Software
2222 Western Trails Suite 106, Austin, Texas 78745
512-441-8354

✧ Request 471 on Reader Service Card ✧

Neural CASE™

More than a shell - a complete neural network application development environment. Train, test, and deliver your application using one product. Supports four network paradigms: BPN, CPN, FLN, and SOM. Object-oriented approach - easily create and configure I/O and network objects using a powerful menu-driven GUI. Weights generated can also be used by the Neurocomputing Library. Introductory price: $199.

Neurocomputing Library

A library of C neural network functions featuring: twelve popular neural network paradigms, multiple networks and paradigms in a program, example programs, and royalty-free distribution of applications. Supports Turbo C 2.0, Turbo C++ V1.0, Microsoft C 5.1, or Zortech C++ (please specify when ordering). $179. Library with source $379.

NeuroSym® Corporation
P.O. Box 980683 • Houston, Texas 77098-0683 • (713) 523-5777

✧ Request 327 on Reader Service Card ✧

PAUWELS CATALOG

A catalog of DOS & OS/2 software on your computer. Cross - referenced on product name, category and editor. Monthly updates.

The PAUWELS CATALOG is distributed as shareware (available on Compuserve - IBMAPP - file PAUWEL. EXE)

A tool for programmers, MIS departments and software dealers.

Pauwels & Partners Software

✧ Request 209 on Reader Service Card ✧

Optimize Your Object Code
Segment Consolidator
An inter-object module optimizer

Convert large code model code to mixed model, delete dead code, convert to register parameter passing, convert to pascal stack clearing convention. No source code modifications required. Output is in ready to link object module format. Supports C language MSC 6.0, Borland C++ 3.0, Assembly generated and runtime library object modules. 30 day money back guarantee. Only 145.00.

Optimite Systems
1000 Singleton Blvd
Dallas, Texas 75212
(214) 745 1301

✧ Request 188 on Reader Service Card ✧

If You Program In C
You Need This Make Utility
Programmer's SUPER-MAINT™

Professional make utility automatically writes your make and response files for you, remembers your command flags, the make file name, and more! Leaves all but about 2K of your memory for compilers to use. 30 day money back guarantee.

Now Version 3 has Smart Library Maintenance, on-the-fly response file building, multiple models built with one command. "Point and shoot" at code files you want in your program, multiple setups, user configurable. Control how your program is built from the command line using simple mnemonic flags. Still *only $55* (plus 3.00 s&h, NY residents add sales tax).

E M M A S O F T

EmmaSoft
PO Box 238
Lansing, NY 14882
Voice: (607) 533-4685
BBS: (607) 533-7072

✧ Request 197 on Reader Service Card ✧

FIGHT PIRACY & PROTECT YOUR PROGRAM $$$'s!

Since 1986, companies worldwide have been choosing Az-Tech security products. If you demand the strongest protection available, why not choose one of these "proven leaders":

- EVERLOCK Copy Protection
- EVERTRAK Software Security
- EVERKEY II "The Lock"

For IBM and Compatibles.
30 day money back guarantee.
Free info and demo disk available.

Az-Tech Software, Inc.
201 East Franklin, Ste. 11
Richmond, MO 64055
(800) 227-0644 (816) 776-2700
FAX (816) 776-8398

✧ Request 245 on Reader Service Card ✧

cascaded. The reason for this is that each individual transform is designed to REJECT and REMOVE information from its output which does not correspond to a gradient in the direction of interest. I noticed in the photo examples that the images had a distinct bias in favor of one gradient over the other. For example, the right edges of dark objects are shown but the left edges of dark objects are missing. This is due to the fact that you have applied the transforms in cascade. The correct use of these transforms dictates that they each be applied to the input image and then the results analyzed.

It should be noted that it is not possible to merely sum the outputs from the eight transforms of the Kirsch and Sobel sets since this would result in values of zero. With these sets however, it is possible to accumulate the absolute value of the transform output. Since the transforms with opposite compass directions create symmetrical output, it is only necessary to use the absolute value of four patterns. The strong negative output of the one direction matrix provides the detection capability previously provided by the opposite transform. The output of the eight Prewitt transforms may be summed to non-zero values but it may be advantageous to use the same feature of symmetry to avoid redundant calculation.

I hope that this comment is not too brief and that you find the information useful. Again thanks, and I look forward to your next article.

Sincerely,

Frank Evans
Intelledex Vision Products

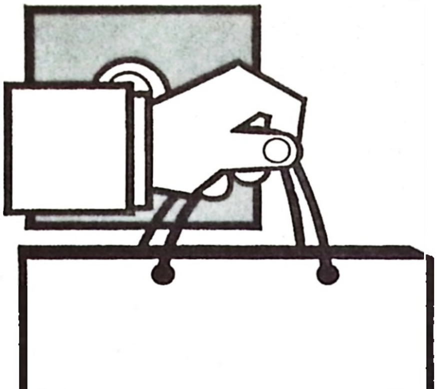
Dwayne Phillips replies: Thanks for the comments.

Gentlemen;

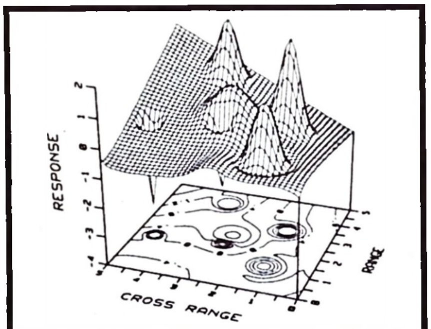
I have just gone through the process of learning how to use TLIB, including 15 minutes on "hold" in a telephone call to Borland. As an ex-professor (of a technical subject), and as a novice C user, I feel it is necessary to make two comments and to editorialize on Mr. Pugh's answer to Mr. Sam LeFevre, of Idaho Falls, as presented in the October 1991 *C Users Journal*.

Mr. LeFevre introduced himself as a "novice Turbo C user," and asked for guidance in setting up his own library of functions. Mr. Pugh correctly advised him to use the TLIB program supplied with Turbo C. But then,

1) went on to hide the really important advice (global variables) in typical and unnecessarily complicated C (for a "novice") programming, and



Programmer's Market



2) omitted the bookkeeping: how does TURBO C access this new library?.

Comment 1: (Probably brought on by some 30 years of teaching) Why not keep it simple? Give the questioner a simple "Hello World" example that stresses library use by itself?

Comment 2: This second comment is much more important and is the result of my telephone call to Borland: Where does one put his library so that Turbo C can find the functions contained in it? The ease with which one can write and save header files (in the same directory as other header files are saved in) would imply that personal libraries could be saved in the library directory. This is not the case. Turbo C is "hard wired" to search only the Borland-supplied libraries. Path(s) to users' libraries must be supplied in make, or project files.

(TLIB does allow you to add functions to (or modify) a current Borland library, but this could prove a dangerous thing to do.)

Thus, when using the very convenient IDE, every program that uses a non-Borland library must be a project, and a project file must be written!

Access to your own library is not explained where it should be (not in the Borland manuals, nor in the two classic Turbo C Bibles by Barkakati), nor can I find it in any of the many Turbo C books that I have compulsively bought. Only H. Schildt, in his *Turbo C, The Complete Reference*, Borland-Osborn/McGraw-Hill, gives this little gem of information. Even here, a confusing (and extraneous?) "T" is introduced.

Indeed, this omission on the part of many authors who glibly write of generating one's own library makes me wonder how many have ever done so?

Owen Gailar
Professor Emeritus, (Purdue University)
345 W. San Carlos
Fresno, CA 93704

I have, and I had to learn all the things you and Ken summarized here. Thanks for the clarifications. — pjp

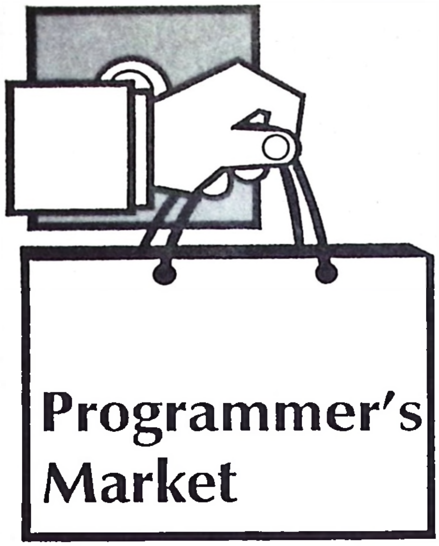
---

Dear Mr. Plauger,

I would like to respond to two items in the December *CUJ*.

First, with regard to C. Skelly's fascinating article, "Creating C++-Like Objects in C." I believe Mr. Skelly made two important points that are worth reiterating.

One, callback functions are the foundation of polymorphic behavior. Understanding this point makes it easier to design flexible systems in C. This point also clarifies a common misconception about C++. The `virtual` keyword is



not free. There is an overhead associated with using virtual functions, which is exactly the overhead of the call by address mechanism.

Two well-constructed C libraries use subject-predicate naming conventions, instead of the more natural seeming predicate-subject. For example, `CircleDraw()` is correct, `DrawCircle()` is incorrect. Having our function names sort on the object name `Circle` allows us to associate the functionality of the `Circle` structure with those functions that begin with that name. This system also has the nice property of sorting well in the symbol tables.

The second point I wish to address refers to Jack Purdum's letter on the *define* versus *declare* problem. I am surprised you don't get it. Stating a thing twice doubles the amount of work needed to maintain it. Any symbol addition in Mr. Purdum's example requires one source code modification, as opposed to the usual two. This problem occurs in a lot of places. Preprocessor tricks, which are all variations on the one given, can save days of hard boring labor in the course of a large project.

Sincerely,

Adam Greissman, President
Hermeios, Inc.
853 Broadway, Suite 1104
New York, NY 10003

I see both advantages and disadvantages in SubjectPredicate naming. I see both advantages and disadvantages in macros that let you declare a data object just once. That makes it hard for me to characterize any of these approaches as "right" or "wrong." —pjp

---

Dear Mr. Plauger:

I would like to request an article discussing the NCEG extensions to Standard C be published. In addition, perhaps several articles discussing issues regarding their use.

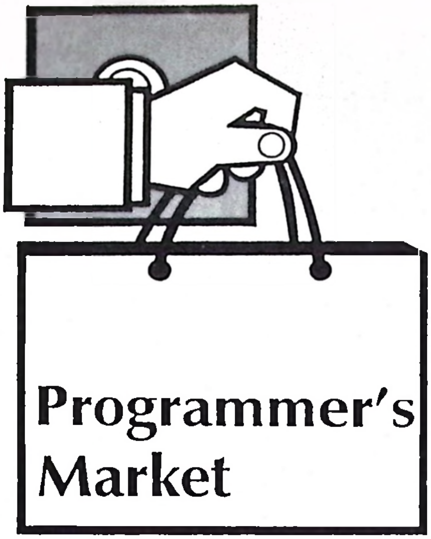
As an avid reader of your column, I would like to know your opinions regarding text formatting packages, especially regarding *nroff/troff* versus the current crop of desktop publishing packages. Would you care to clarify it in an upcoming column?

Thanks for your help.

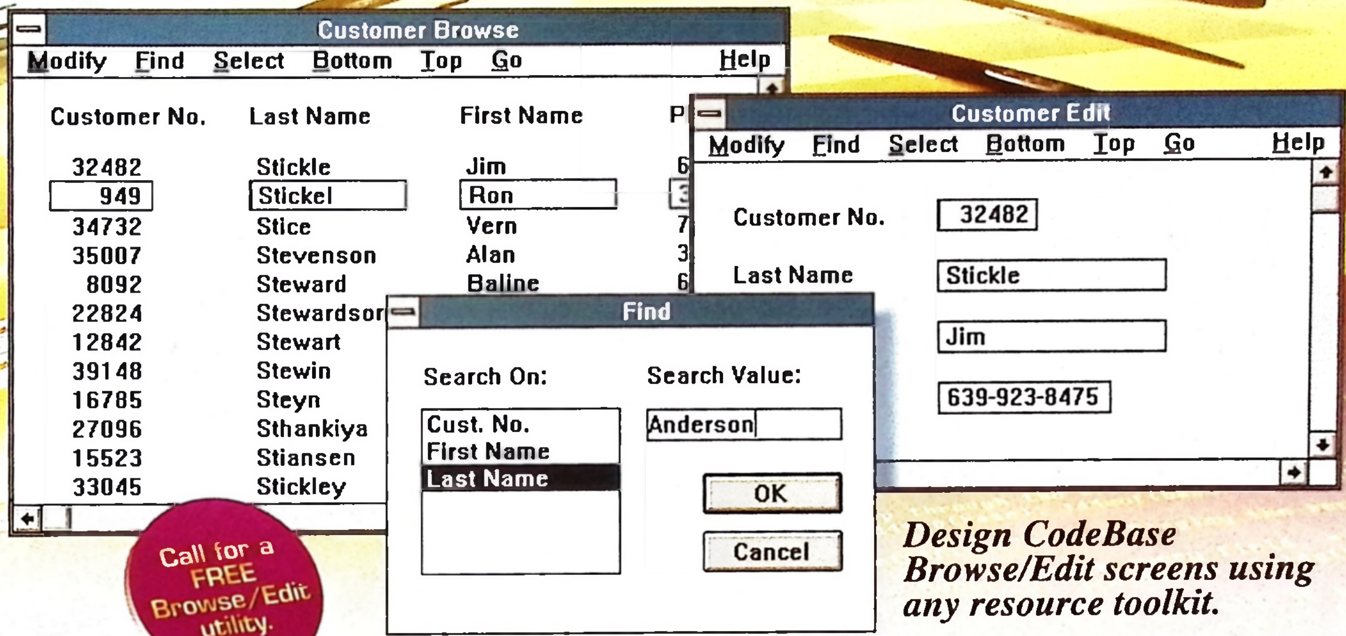
Sincerely,

Wen J. Chen
143-25 41st Avenue, #340
Flushing, NY 11355

I haven't discussed NCEG issues in my column yet for two reasons. First, I still have a few more installments on the Standard C library. Second, the NCEG folk are still settling down on the content of their technical report. I fully expect to devote some space to their work at the appropriate time. — pjp



Access dBASE IV and FoxPro files from C or C++



Call for a FREE Browse/Edit utility.

Design CodeBase Browse/Edit screens using any resource toolkit.

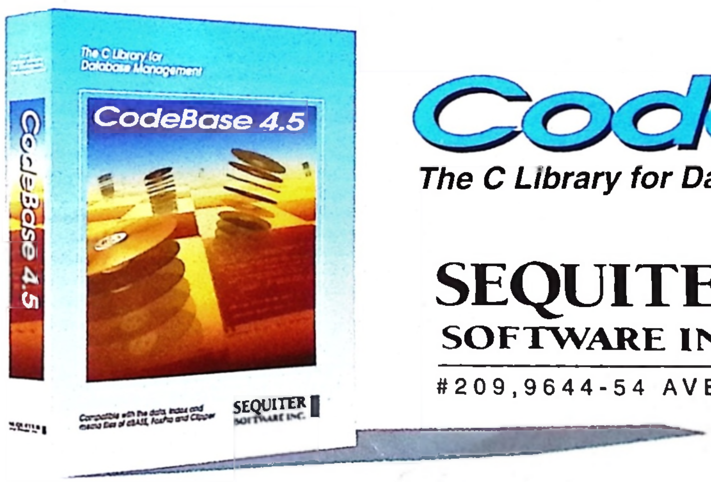
- **Multi User**
- **Portable (DOS, Unix, ...)**
- **Royalty Free DLL**
- **C++ interface included**

Use CodeBase 4.5 from **Visual Basic** or **Turbo Pascal for Windows.**

Use the super-fast, super-small **FoxPro 2.0** CDX or the Clipper NTX index files.

" Our product was too slow under FoxPro 2.0, so we rewrote it in C using CodeBase. Now it is incredibly fast."

Jeff Reed, DCS Computer Services



CodeBase 4.5

The C Library for DataBase Management

SEQUITER SOFTWARE INC.

TEL. 403•437•2410
FAX 403•436•2999
Europe 33.20.24.20.14

#209,9644-54 AVE., EDMONTON, AB, CANADA T6E-5V1

✧ Request 166 on Reader Service Card ✧